

Google Android官方培训教程

中文版

目錄

序言 1.1
Android入门基础：从这里开始 1.2
建立第一个App 1.2.1
创建Android项目 1.2.1.1
执行Android程序 1.2.1.2
建立简单的用户界面 1.2.1.3
启动其他的Activity 1.2.1.4
添加ActionBar 1.2.2
建立ActionBar 1.2.2.1
添加Action按钮 1.2.2.2
自定义ActionBar的风格 1.2.2.3
ActionBar的覆盖层叠 1.2.2.4
兼容不同的设备 1.2.3
适配不同的语言 1.2.3.1
适配不同的屏幕 1.2.3.2
适配不同的系统版本 1.2.3.3
管理Activity的生命周期 1.2.4
启动与销毁Activity 1.2.4.1
暂停与恢复Activity 1.2.4.2
停止与重启Activity 1.2.4.3
重新创建Activity 1.2.4.4
使用Fragment建立动态的UI 1.2.5
创建一个Fragment 1.2.5.1
建立灵活动态的UI 1.2.5.2
Fragments之间的交互 1.2.5.3
数据保存 1.2.6
保存到Preference 1.2.6.1
保存到文件 1.2.6.2
保存到数据库 1.2.6.3
与其他应用的交互 1.2.7

- Intent的发送 1.2.7.1
- 接收Activity返回的结果 1.2.7.2
- Intent过滤 1.2.7.3

Android分享操作 1.3

- 分享简单的数据 1.3.1
  - 给其他App发送简单的数据 1.3.1.1
  - 接收从其他App返回的数据 1.3.1.2
  - 给ActionBar增加分享功能 1.3.1.3
- 分享文件 1.3.2
  - 建立文件分享 1.3.2.1
  - 分享文件 1.3.2.2
  - 请求分享一个文件 1.3.2.3
  - 获取文件信息 1.3.2.4
- 使用NFC分享文件 1.3.3
  - 发送文件给其他设备 1.3.3.1
  - 接收其他设备的文件 1.3.3.2

Android多媒体 1.4

- 管理音频播放 1.4.1
  - 控制音量与音频播放 1.4.1.1
  - 管理音频焦点 1.4.1.2
  - 兼容音频输出设备 1.4.1.3
- 拍照 1.4.2
  - 简单的拍照 1.4.2.1
  - 简单的录像 1.4.2.2
  - 控制相机硬件 1.4.2.3
- 打印 1.4.3
  - 打印照片 1.4.3.1
  - 打印HTML文档 1.4.3.2
  - 打印自定义文档 1.4.3.3

Android图像与动画 1.5

- 高效显示Bitmap 1.5.1
  - 高效加载大图 1.5.1.1
  - 非UI线程处理Bitmap 1.5.1.2
  - 缓存Bitmap 1.5.1.3

管理Bitmap的内存 1.5.1.4
在UI上显示Bitmap 1.5.1.5
使用OpenGL ES显示图像 1.5.2
建立OpenGL ES的环境 1.5.2.1
定义Shapes 1.5.2.2
绘制Shapes 1.5.2.3
运用投影与相机视图 1.5.2.4
添加移动 1.5.2.5
响应触摸事件 1.5.2.6
添加动画 1.5.3
View间渐变 1.5.3.1
使用ViewPager实现屏幕滑动 1.5.3.2
展示Card翻转动画 1.5.3.3
缩放View 1.5.3.4
布局变更动画 1.5.3.5
Android网络连接与云服务 1.6
无线连接设备 1.6.1
使用网络服务发现 1.6.1.1
使用WiFi建立P2P连接 1.6.1.2
使用WiFi P2P服务 1.6.1.3
执行网络操作 1.6.2
连接到网络 1.6.2.1
管理网络的使用情况 1.6.2.2
解析XML数据 1.6.2.3
传输数据时避免消耗大量电量 1.6.3
优化下载以高效地访问网络 1.6.3.1
最小化定期更新造成的影响 1.6.3.2
重复的下载是冗余的 1.6.3.3
根据网络连接类型来调整下载模式 1.6.3.4
云同步 1.6.4
使用备份API 1.6.4.1
使用Google Cloud Messaging 1.6.4.2
解决云同步的保存冲突 1.6.5

使用Sync Adapter传输数据 1.6.6
创建Stub授权器 1.6.6.1
创建Stub Content Provider 1.6.6.2
创建Sync Adpater 1.6.6.3
执行Sync Adpater 1.6.6.4
使用Volley执行网络数据传输 1.6.7
发送简单的网络请求 1.6.7.1
建立请求队列 1.6.7.2
创建标准的网络请求 1.6.7.3
实现自定义的网络请求 1.6.7.4
Android联系人与位置信息 1.7
Android联系人信息 1.7.1
获取联系人列表 1.7.1.1
获取联系人详情 1.7.1.2
使用Intents修改联系人信息 1.7.1.3
显示联系人头像 1.7.1.4
Android位置信息 1.7.2
获取最后可知位置 1.7.2.1
获取位置更新 1.7.2.2
显示位置地址 1.7.2.3
创建和监视地理围栏 1.7.2.4
Android可穿戴应用 1.8
赋予Notification可穿戴特性 1.8.1
创建Notification 1.8.1.1
在Notifcation中接收语音输入 1.8.1.2
为Notification添加显示页面 1.8.1.3
以Stack的方式显示Notifications 1.8.1.4
创建可穿戴的应用 1.8.2
创建并运行可穿戴应用 1.8.2.1
创建自定义的布局 1.8.2.2
添加语音功能 1.8.2.3
打包可穿戴应用 1.8.2.4
通过蓝牙进行调试 1.8.2.5
创建自定义的UI 1.8.3

定义Layouts 1.8.3.1
创建Card 1.8.3.2
创建List 1.8.3.3
创建2D Picker 1.8.3.4
创建确认界面 1.8.3.5
退出全屏的Activity 1.8.3.6
发送并同步数据 1.8.4
访问可穿戴数据层 1.8.4.1
同步数据单元 1.8.4.2
传输资源 1.8.4.3
发送与接收消息 1.8.4.4
处理数据层的事件 1.8.4.5
创建表盘 1.8.5
设计表盘 1.8.5.1
构建表盘服务 1.8.5.2
绘制表盘 1.8.5.3
在表盘上显示信息 1.8.5.4
提供配置 Activity 1.8.5.5
定位常见的问题 1.8.5.6
优化性能和电池使用时间 1.8.5.7
位置检测 1.8.6
Android TV应用 1.9
创建TV应用 1.9.1
创建TV应用的第一步 1.9.1.1
处理TV硬件部分 1.9.1.2
创建TV的布局文件 1.9.1.3
创建TV的导航栏 1.9.1.4
创建TV播放应用 1.9.2
创建目录浏览器 1.9.2.1
提供一个Card视图 1.9.2.2
创建详情页 1.9.2.3
显示正在播放卡片 1.9.2.4
帮助用户在TV上探索内容 1.9.3

TV上的推荐内容 1.9.3.1
使得TV App能够被搜索 1.9.3.2
使用TV应用进行搜索 1.9.3.3
创建TV游戏应用 1.9.4
创建TV直播应用 1.9.5
TV Apps Checklist 1.9.6
Android企业级应用 1.10
Ensuring Compatibility with Managed Profiles 1.10.1
Implementing App Restrictions 1.10.2
Building a Work Policy Controller 1.10.3
Android交互设计 1.11
设计高效的导航 1.11.1
规划屏幕界面与他们之间的关系 1.11.1.1
为多种大小的屏幕进行规划 1.11.1.2
提供向下和横向导航 1.11.1.3
提供向上和历史导航 1.11.1.4
综合：设计样例 App 1.11.1.5
实现高效的导航 1.11.2
使用Tabs创建Swipe视图 1.11.2.1
创建抽屉导航 1.11.2.2
提供向上的导航 1.11.2.3
提供向后的导航 1.11.2.4
实现向下的导航 1.11.2.5
通知提示用户 1.11.3
建立Notification 1.11.3.1
当启动Activity时保留导航 1.11.3.2
更新Notification 1.11.3.3
使用BigView风格 1.11.3.4
显示Notification进度 1.11.3.5
增加搜索功能 1.11.4
建立搜索界面 1.11.4.1
保存并搜索数据 1.11.4.2
保持向下兼容 1.11.4.3
使得你的App内容可被Google搜索 1.11.5

为App内容开启深度链接 1.11.5.1
为索引指定App内容 1.11.5.2
Android界面设计 1.12
为多屏幕设计 1.12.1
兼容不同的屏幕大小 1.12.1.1
兼容不同的屏幕密度 1.12.1.2
实现可适应的UI 1.12.1.3
创建自定义View 1.12.2
创建自定义的View类 1.12.2.1
实现自定义View的绘制 1.12.2.2
使得View可交互 1.12.2.3
优化自定义View 1.12.2.4
创建向后兼容的UI 1.12.3
抽象新的APIs 1.12.3.1
代理至新的APIs 1.12.3.2
使用旧的APIs实现新API的效果 1.12.3.3
使用版本敏感的组件 1.12.3.4
实现辅助功能 1.12.4
开发辅助程序 1.12.4.1
开发辅助服务 1.12.4.2
管理系统UI 1.12.5
淡化系统Bar 1.12.5.1
隐藏系统Bar 1.12.5.2
隐藏导航Bar 1.12.5.3
全屏沉浸式应用 1.12.5.4
响应UI可见性的变化 1.12.5.5
创建使用Material Design的应用 1.12.6
开始使用Material Design 1.12.6.1
使用Material的主题 1.12.6.2
创建Lists与Cards 1.12.6.3
定义Shadows与Clipping视图 1.12.6.4
使用Drawables 1.12.6.5
自定义动画 1.12.6.6

维护兼容性 1.12.6.7
Android用户输入 1.13
使用触摸手势 1.13.1
检测常用的手势 1.13.1.1
跟踪手势移动 1.13.1.2
滚动手势动画 1.13.1.3
处理多点触控手势 1.13.1.4
拖拽与缩放 1.13.1.5
管理ViewGroup中的触摸事件 1.13.1.6
处理键盘输入 1.13.2
指定输入法类型 1.13.2.1
处理输入法可见性 1.13.2.2
支持键盘导航 1.13.2.3
处理按键动作 1.13.2.4
支持游戏控制器 1.13.3
处理控制器输入动作 1.13.3.1
在不同的 Android 系统版本支持控制器 1.13.3.2
支持多个控制器 1.13.3.3
Android后台任务 1.14
在IntentService中执行后台任务 1.14.1
创建IntentService 1.14.1.1
发送工作任务到IntentService 1.14.1.2
报告后台任务执行状态 1.14.1.3
使用CursorLoader在后台加载数据 1.14.2
使用CursorLoader执行查询任务 1.14.2.1
处理CursorLoader查询的结果 1.14.2.2
管理设备的唤醒状态 1.14.3
保持设备的唤醒 1.14.3.1
制定重复定时的任务 1.14.3.2
Android性能优化 1.15
管理应用的内存 1.15.1
代码性能优化建议 1.15.2
提升Layout的性能 1.15.3
优化layout的层级 1.15.3.1

使用include标签重用layouts	1.15.3.2
按需加载视图	1.15.3.3
使得ListView滑动顺畅	1.15.3.4
优化电池寿命	1.15.4
监测电量与充电状态	1.15.4.1
判断与监测Docking状态	1.15.4.2
判断与监测网络连接状态	1.15.4.3
根据需要操作Broadcast接受者	1.15.4.4
多线程操作	1.15.5
在一个线程中执行一段特定的代码	1.15.5.1
为多线程创建线程池	1.15.5.2
启动与停止线程池中的线程	1.15.5.3
与UI线程通信	1.15.5.4
避免出现程序无响应ANR	1.15.6
JNI使用指南	1.15.7
优化多核处理器(SMP)下的Android程序	1.15.8
Android安全与隐私	1.16
Security Tips	1.16.1
使用HTTPS与SSL	1.16.2
为防止SSL漏洞而更新Security	1.16.3
使用设备管理条例增强安全性	1.16.4
Android测试程序	1.17
测试你的Activity	1.17.1
建立测试环境	1.17.1.1
创建与执行测试用例	1.17.1.2
测试UI组件	1.17.1.3
创建单元测试	1.17.1.4
创建功能测试	1.17.1.5

Android官方培训课程中文版(v0.9.7)



Google Android团队在2012年的时候开设了**Android Training**板块 - http://developer.android.com/training/index.html，这些课程是学习Android应用开发的绝佳资料。我们通过Github发起开源协作翻译的项目，完成中文版的输出，欢迎大家传阅学习！ 文章中难免会有很多写的不对不好的地方，欢迎读者加入此协作项目，进行纠错，为完善这份教程贡献一点力量!

Github托管主页

https://github.com/kesenhoo/android-training-course-in-chinese

请读者点击Star进行关注并支持！

在线阅读

http://hukai.me/android-training-course-in-chinese/index.html

更新记录

- v0.9.7 - 2016/11/04
- v0.9.6 - 2016/05/22
- v0.9.5 - 2015/12/15
- v0.9.4 - 2015/06/11
- v0.9.3 - 2015/05/18
- v0.9.2 - 2015/03/30

- v0.9.1 - 2015/03/14
- v0.9.0 - 2015/03/09
- v0.8.0 - 2015/02/12
- v0.7.0 - 2014/11/30
- v0.6.0 - 2014/11/02
- v0.5.0 - 2014/10/18
- v0.4.0 - 2014/09/11
- v0.3.0 - 2014/08/31
- v0.2.0 - 2014/08/14
- v0.1.0 - 2014/08/05

参与方式

你可以选择以下的方式帮忙修改纠正这份教程（推荐使用方法1）：

1. 通过在线阅读课程的页面，找到Github仓库对应的章节文件，直接在线编辑修改提交即可。
2. 在线阅读的文章底部留言，提出问题与修改意见，我会抽时间及时处理。
3. 写邮件给发起人：胡凯，邮箱是kesenhoo at gmail.com，邮件内容注明需要纠正的章节段落位置，并给出纠正的建议。

致谢

发起这个项目之后，得到很多人的支持，有经验丰富的Android开发者，也有刚接触Android的爱好者。他们有些已经上班，有些还是学生，有些在国内，还有的在国外！感谢所有参与或者关注这个项目的小伙伴!

下面是参与翻译的小伙伴(Github ID按照课程结构排序)：

0	1	2
@yuanfentiank789	@vincent4j	@Lin-H
@kesenhoo	@fastcome1985	@jdneo
@XizhiXu	@naizhengtan	@spencer198711
@penkzhou	@wangyachen	@wly2014
@fastcome1985	@riverfeng	@xrayzh
@K0ST	@Andrwyw	@zhaochunqi
@lltowq	@allenlsy	@AllenZheng1991
@pedant	@craftsmanBai	@huanglizhuo
@Roya	@awong1900	@dupengwei
0:10	1:10	2:10

@发起人:胡凯，博客：http://hukai.me，Github：https://github.com/kesenhoo，微博：http://weibo.com/kesenhoo

还有众多参与纠错校正的同学名字就不一一列举了，谢谢所有关注这个项目的小伙伴！特别感谢安卓巴士社区，爱开发社区，码农周刊对项目的宣传！

License

Android入门基础：从这里开始

> 编写:kesenhoo - 原文:http://developer.android.com/training/index.html

欢迎来到为Android开发者准备的培训项目。在这里你会找到一系列的课程，这些课程会演示你如何使用可重用的代码来完成特定的任务。所有的课程分为若干不同的小组。你可以通过左边的导航来查看。

第1章节："从这里开始"，教你Android应用开发的最基本的知识。如果你是一个Android应用开发的新手，你应该按照顺序学习完下面的课程：

建立你的第一个App(Building Your First App)

在你安装Android SDK之后，从这节课开始学习Android应用开发的基础知识。

兼容不同的设备(Supporting Different Devices)

学习给应用提供可选择的资源文件来实现如何使用一个APK来使得你的应用能够在不同的设备上获取到最佳的用户体验。

使用Fragment建立动态的UI(Building a Dynamic UI with Fragments)

学习如何为你的应用建立一套足够灵活的UI，这套UI能够在大屏幕的设备上显示多个UI组件，在小屏幕的设备上呈现紧凑的UI组件。这使得你能够为手机与平板只建立同一个APK。

数据保存(Saving Data)

学习如何在设备上保存数据。无论这些数据是临时的文件，应用下载的资源，用户的多媒体数据，结构化的数据还是其他。

与其他应用的交互(Interacting with Other Apps)

学习如何利用其他已经存在应用的既有功能来执行更进一步的用户任务。例如拍照或者在地图上查看某个地址。

使用系统权限(Working with System Permissions)

学习声明你的app需要访问在它"沙箱"之外的功能和资源，以及如何在运行时申请这些特权。

建立第一个App

> 编写:yuanfentiank789 - 原文:http://developer.android.com/training/basics/firstapp/index.html

欢迎开始Android应用开发之旅！

本章节我们将学习如何建立我们的第一个Android应用程序。我们将学到如何使用Android Studio创建一个Android项目并运行该应用程序的可调试版本。此外，我们还将学习到一些Android应用程序设计的基础知识，包括如何创建一个简单的用户界面，以及处理用户输入。

开始本章节学习之前，我们要确保已经安装了开发环境。我们需要：

1 下载并安装Android Studio.

2 使用SDK Manager（可以设置g.cn:80作为SDK代理，实现免翻墙更新SDK）下载最新的SDK tools和platforms。

> **Note**：虽然这一系列的培训课程中的大多数章节都预期你会使用Android Studio来进行开发，但某些开发操作还是可以通过SDK tools中提供的命令来实现的。

本章节通过向导的方式来逐步创建一个小型的Android应用，通过这些步骤来教给我们一些Android开发的基本概念，因此你很有必要按照教程的步骤来学习操作。

开始学习

创建Android项目

> 编写:yuanfentiank789 - 原文:http://developer.android.com/training/basics/firstapp/creating-project.html

一个Android项目包含了所有构成Android应用的源代码文件。

本小节介绍如何使用Android Studio或者是SDK Tools中的命令行来创建一个新的项目。

> **Note**：在此之前，我们应该已经安装了Android SDK，如果使用Android Studio开发，应该确保已经安装了Android Studio。否则，请先阅读 Installing the Android SDK按照向导完成安装步骤。

使用Android Studio创建项目

1. 使用Android Studio创建Android项目，启动Android Studio。

- 如果我们还没有用Android Studio打开项目，会看到欢迎页，点击**Start a new Android Studio project**。
- 如果已经用Android Studio打开了项目，点击菜单中的File，选择New Project来创建一个新的项目。

2.在弹出的窗口（**Configure your new project**）中填入内容，点击**Next**。按照如下的值进行填写会使得后续的操作步骤不容易出差错。

- **Application Name** 此处填写想呈现给用户的应用名称，此处我们使用“My First App”。
- **Company domain** 包名限定符，Android Studio会将这个限定符应用于每个新建的Android项目，此处使用"example.com"。
- **Package Name** 是应用的包命名空间（同Java的包的概念），该包名在同一Android系统上所有已安装的应用中具有唯一性，由包名限定符而定。
- **Project location** 操作系统存放项目的路径。

3. 在**Select the form factors your app will run on**窗口勾选**Phone and Tablet**。

4. **Minimum SDK**, 选择 **API 15: Android 4.0.3 (IceCreamSandwich)**. Minimum Required SDK表示我们的应用支持的最低Android版本，为了支持尽可能多的设备，我们应该设置为能支持你应用核心功能的最低API版本。如果某些非核心功能仅在较高版本的API支持，你可以只在支持这些功能的版本上开启它们(参考兼容不同的系统版本),此处采用默认值即可。

5. 不要勾选其他选项 (TV, Wear, and Glass) ，点击 **Next.**

6. 在**Add an activity to Mobile** 窗口选择**Empty Activity**，点击 **Next.**

7. 在**Customize the Activity** 窗口保持默认设置。

8. 点击**Finish**完成创建。

刚创建的Android项目是一个基础的Hello World项目，包含一些默认文件，我们花一点时间看看最重要的部分：

`app/res/layout/activity_main.xml`

这是刚才用Android Studio创建项目时新建的Activity对应的xml布局文件，按照创建新项目的流程，Android Studio会同时展示这个文件的文本视图和图形化预览视图，该文件包含一些默认设置和一个显示内容为“Hello world!”的TextView元素。

`app/java/com.example.myfirstapp/MainActivity.java`

用Android Studio创建新项目完成后，可在Android Studio看到该文件对应的选项卡，选中该选项卡，可以看到刚创建的Activity类的定义。编译并运行该项目后，Activity启动并加载布局文件activity_main.xml，显示一条文本："Hello world!"

`app/manifest/AndroidManifest.xml`

manifest文件描述了项目的基本特征并列出了组成应用的各个组件，接下来的学习会更深入了解这个文件并添加更多组件到该文件中。

`app/build.gradle`

Android Studio使用Gradle 编译运行Android工程. 工程的每个模块以及整个工程都有一个build.gradle文件。通常你只需要关注模块的build.gradle文件，该文件存放编译依赖设置，包括defaultConfig设置：

- compiledSdkVersion 是我们的应用将要编译的目标Android版本，此处默认为你的SDK已安装的最新Android版本(目前应该是4.1或更高版本，如果你没有安装一个可用Android版本，就要先用SDK Manager来完成安装)，我们仍然可以使用较老的版本编译项目，但把该值设为最新版本，可以使用Android的最新特性，同时可以在最新的设备上优化应用来提高用户体验。
- applicationId 创建新项目时指定的包名。
- minSdkVersion 创建项目时指定的最低SDK版本，是新建应用支持的最低SDK版本。
- targetSdkVersion 表示你测试过你的应用支持的最高Android版本(同样用API level表示). 当Android发布最新版本后，我们应该在最新版本的Android测试自己的应用同时更新target sdk到Android最新版本，以便充分利用Android新版本的特性。更多知识，请阅读Supporting Different Platform Versions。

更多关于Gradle的知识请阅读Building Your Project with Gradle

注意/res目录下也包含了resources资源：

`drawable<density>/`

存放各种densities图像的文件夹，mdpi，hdpi等，这里能够找到应用运行时的图标文件ic_launcher.png

`layout/`

存放用户界面文件，如前边提到的activity_my.xml，描述了MyActivity对应的用户界面。

`menu/`

存放应用里定义菜单项的文件。

`values/`

存放其他xml资源文件，如string，color定义。string.xml定义了运行应用时显示的文本"Hello world!"

要运行这个APP，继续下个小节的学习。

使用命令行创建项目

如果没有使用Android Studio开发Android项目，我们可以在命令行使用SDK提供的tools来创建一个Android项目。

1. 打开命令行切换到SDK根目录下；

2. 执行:

```
tools/android list targets
```

会在屏幕上打印出我们所有的Android SDK中下载好的可用Android platforms，找想要创建项目的目标platform，记录该platform对应的Id，推荐使用最新的platform。我们仍可以使自己的应用支持较老版本的platform，但设置为最新版本允许我们为最新的Android设备优化我们的应用。 如果没有看到任何可用的platform，我们需要使用Android SDK Manager完成下载安装，参见 Adding Platforms and Packages。

3. 执行：

```
android create project --target <target-id> --name MyFirstApp \
--path <path-to-workspace>/MyFirstApp --activity MyActivity \
--package com.example.myfirstapp
```

替换 `<target-id>` 为上一步记录好的Id，替换 `<path-to-workspace>` 为我们想要保存项目的路径。

> **Tip**:把 `platform-tools/` 和 `tools/` 添加到环境变量 `PATH` ，开发更方便。

到此为止，我们的Android项目已经是一个基本的“Hello World”程序，包含了一些默认的文件。要运行它，继续下个小节的学习。

执行Android程序

> 编写:yuanfentiank789 - 原文:http://developer.android.com/training/basics/firstapp/running-app.html

通过上一节课创建了一个Android的Hello World项目，项目默认包含一系列源文件，它让我们可以立即运行应用程序。

如何运行Android应用取决于两件事情：是否有一个Android设备和是否正在使用Android Studio开发程序。本节课将会教使用Android Studio和命令行两种方式在真实的android设备或者android模拟器上安装并且运行应用。

在真实设备上运行

如果有一个真实的Android设备，以下的步骤可以使我们在自己的设备上安装和运行应用程序：

手机设置

1. 把设备用USB线连接到计算机上。如果是在windows系统上进行开发的，你可能还需要安装你设备对应的USB驱动，详见OEM USB Drivers 文档。
2. 开启设备上的**USB**调试选项。
    - 在大部分运行Andriod3.2或更老版本系统的设备上，这个选项位于“设置>应用程序>开发选项”里。
    - 在Andriod 4.0或更新版本中，这个选项在“设置>开发人员选项”里。

> **Note:** 从Android4.2开始，开发人员选项在默认情况下是隐藏的，想让它可见，可以去设置>关于手机（或者关于设备)点击版本号七次。再返回就能找到开发人员选项了。

从Android Studio运行程序

1. 选择项目的一个文件，点击工具栏里的**Run**按钮。
2. **Choose Device**窗口出现时，选择**Choose a running device**单选框，点击**OK**。

Android Studio 会把应用程序安装到我们的设备中并启动应用程序。

从命令行安装运行应用程序

打开命令行并切换当前目录到Andriod项目的根目录，在debug模式下使用Gradle编译项目，使用gradle脚本执行assembleDebug编译项目，执行后会在build/目录下生成MyFirstApp-debug.apk。

Windows操作系统下，执行：

```
gradlew.bat assembleDebug
```

Mac OS或Linux系统下：

```
$ chmod +x gradlew
$ ./gradlew assembleDebug
```

编译完成后在app/目录生成apk。

> **Note:** chmod命令是给gradlew增加执行权限，只需要执行一次。

确保 Android SDK里的 `platform-tools/` 路径已经添加到环境变量 `PATH` 中，执行：

```
adb install bin/MyFirstApp-debug.apk
```

在我们的Android设备中找到 MyFirstActivity，点击打开。

在模拟器上运行

无论是使用 Android Studio 还是命令行，在模拟器中运行程序首先要创建一个 Android Virtual Device (AVD)。AVD 是对 Android 模拟器的配置，可以让我们模拟不同的设备。

创建一个 AVD:

1. 启动 Android Virtual Device Manager（AVD Manager）的两种方式：

```
* 用Android Studio, **Tools > Android > AVD Manager**,或者点击工具栏里面Android Virtual D
evice Manager![image](avd-manager-studio.png)；
* 在命令行窗口中，把当前目录切换到`<sdk>/tools/` 后执行：
```

```
android avd
```



2. 在AVD Manager 面板中，点击**Create Virtual Device.**

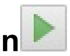
3. 在Select Hardware窗口，选择一个设备，比如 Nexus 6，点击**Next**。

4. 选择列出的合适系统镜像.

5. 校验模拟器配置，点击**Finish**。

更多AVD的知识请阅读Managing AVDs with AVD Manager.

从**Android Studio**运行程序：

1. 在Android Studio选择要运行的项目，从工具栏选择**Run**；

2. **Choose Device**窗口出现时，选择**Launch emulator**单选框；

3. 从 **Android virtual device**下拉菜单选择创建好的模拟器，点击**OK**；

模拟器启动需要几分钟的时间，启动完成后，解锁即可看到程序已经运行到模拟器屏幕上了。

从命令行安装运行应用程序

1. 用命令行编译应用，生成位于app/build/outputs/apk/的apk；

2. 确认platform-tools/ 已添加到PATH环境变量；

3. 执行如下命令：

```
adb install app/build/outputs/apk/MyFirstApp-debug.apk
```

4. 在模拟器上找到MyFirstApp，并运行。

以上就是创建并在设备上运行一个应用的全部过程！想要开始开发，点击next lesson。

建立简单的用户界面

> 编写：crazypudding - 原文：http://developer.android.com/training/basics/firstapp/building-ui.html

在本小节里，我们将学习使用Android Studio布局编辑器创建一个带有文本输入框和按钮的界面。下一节课将学会使 APP 对按钮做出响应——按钮被按下时，文本框里的内容被发送到另外一个 Activity。

Android 的图形用户界面由多个 视图（View）和 布局（ViewGroup）构建而成。View 是通用的 UI 窗体小组件，如：按钮（Button）、文本框（Text field）；而 ViewGroup 则是用来控制子视图如何显示在屏幕上的不可见的容器，如：网格部件（grid）、垂直列表部件（vertical list）。

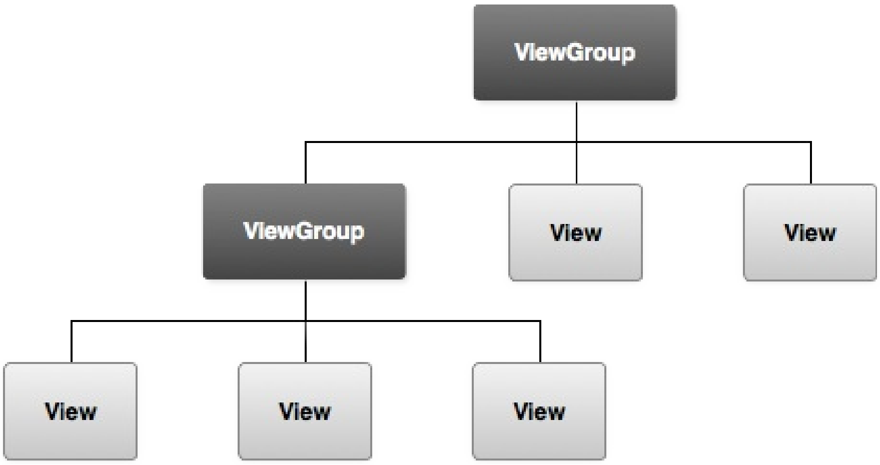


图 **1** 关于 ViewGroup 对象如何组织布局分支和包含其他 View 对象。

Android 提供了一系列对应于 View 和 ViewGroup 子类的 XML 标签，大多数情况下，我们都会使用 XML 来定义自己的UI。不过这节课中我们不会练习 XML 语法，而是练习使用 Android Studio 的布局编辑器来创建布局，布局编辑器通过拖放 View 的方式可以更容易的创建一个布局。

打开布局编辑器

> 注意： 下面的内容都假定我们使用Android Studio 2.3或2.3以上的版本并且通过之前的课程的内容创建了一个Android项目。

开始之前，按照如下步骤设置好工作台：

1.在Android Studio 的 Project 面板中，打开文件 `app/res/layout/activity_main.xml` 。

2.为布局编辑器留出更多空间，通过选择 `View > Tool Windows > Project` 来关闭 **Project** 面板（或者点击 Android Studio 左侧的 按钮）。

3.如果编辑器显示的是 XML 源码，点击左下角 **Design** 标签切换到 Design 模式。

4.点击 **Show Blueprint** 只显示蓝图布局。

5.在布局中显示 Constraints。将鼠标放在工具栏中 按钮上会看到提示： **Hide Constraints**（当前为显示 Constraints）。

6.关闭自动连接功能。将鼠标放在工具栏中 按钮上会看到提示： **Turn On Autoconnect**（当前为关闭状态）。

7.点击工具栏中 **Default Margins** 8 按钮并选择 **16**（稍后仍可以单独为每个 View 调整间距）。

8.点击工具栏中 **Device in Editor** 按钮并选择 **Pixel XL**。

以上操作完成后，Android Studio窗口应该如下图2所示

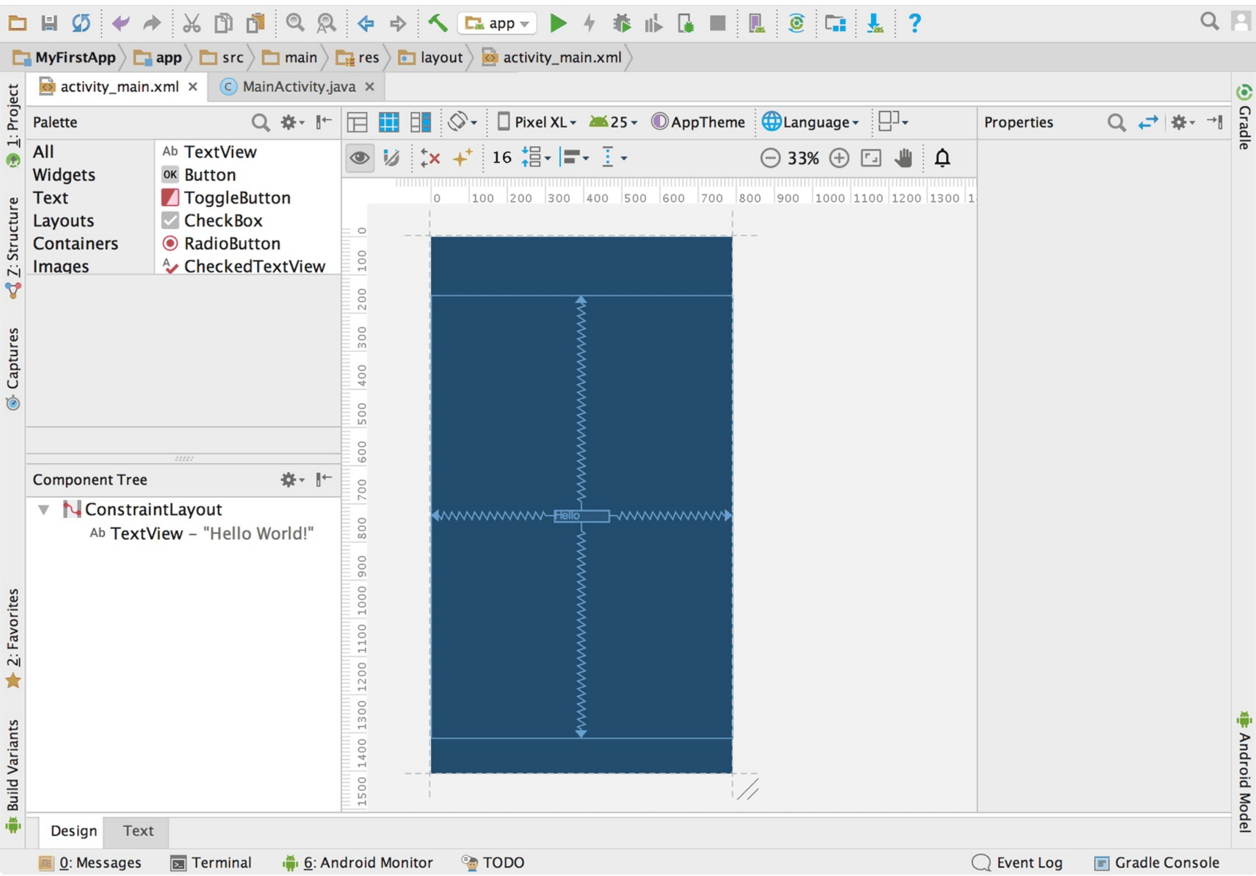


图 **2.** 显示 `activity_main.xml` 的布局编辑器

左下角的 **Component Tree** 面板显示的是当前布局中所有 View 的层级结构。本例中，根 View 是一个 `ConstraintLayout`，其中只包含一个 `TextView` 对象。

`ConstraintLayout` 根据每个 View 和与它平级的兄弟 View 以及父布局之间的约束来确定它的位置。通过这个方法，我们可以创建简单或者复杂但是层级结构扁平化的布局。这样一来，就避免了嵌套布局的出现（如图1展示的那样，一个 ViewGroup 嵌套另一个 ViewGroup），缩短了绘制 UI 的时间。

例如，我们可以这样创建布局（如图3）：

- View A 距父布局顶部16dp
- View A 距父布局左边缘16dp
- View B 距 View A 右边16dp
- View B 与 View A 顶部对齐

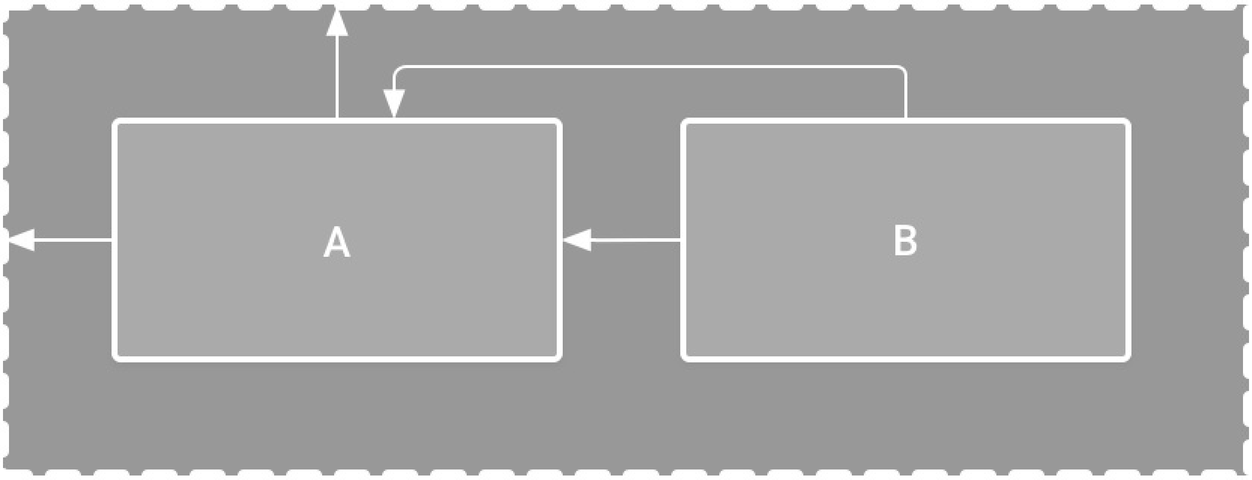


图 **3.** `ConstraintLayout` 中两个 View 的位置

在本节后面的部分中，我们将实际建立一个类似的布局。

添加一个文本框

1.首先，要删除布局中已经存在的 View，在 **Component Tree** 面板中选中并删除 **TextView**。

2.在左侧 **Palette** 面板的左半部分窗格中选中 **Text** 分类，从右半部分窗格中拖出 **Plain Text** 并把它放到编辑器中靠近布局顶部的地方。这是一个可以输入纯文本的 EditText

3.点击编辑器中的 View。可以看到，在每个角上都有一个方形的锚点，这是用来控制 View 的大小的；在每条边中间都有一个圆形锚点，这是用来添加约束的。

为了更准确的控制这些锚点，可以通过工具栏中的缩放按钮来缩放虚拟 UI 界面。

4.按住 View 顶部的圆形锚点，将它拖动到父布局顶部，直到有吸附效果时放开。可以看到 View 和父布局顶部之间出现一条带箭头的细线，这就是一个约束——它指定 View 距离父布局顶部16dp（因为刚刚设置的默认值是16dp）。

5.同样的，在 View 的左边和父布局的左边缘创建一个约束。

最终的效果应该如图4所示。

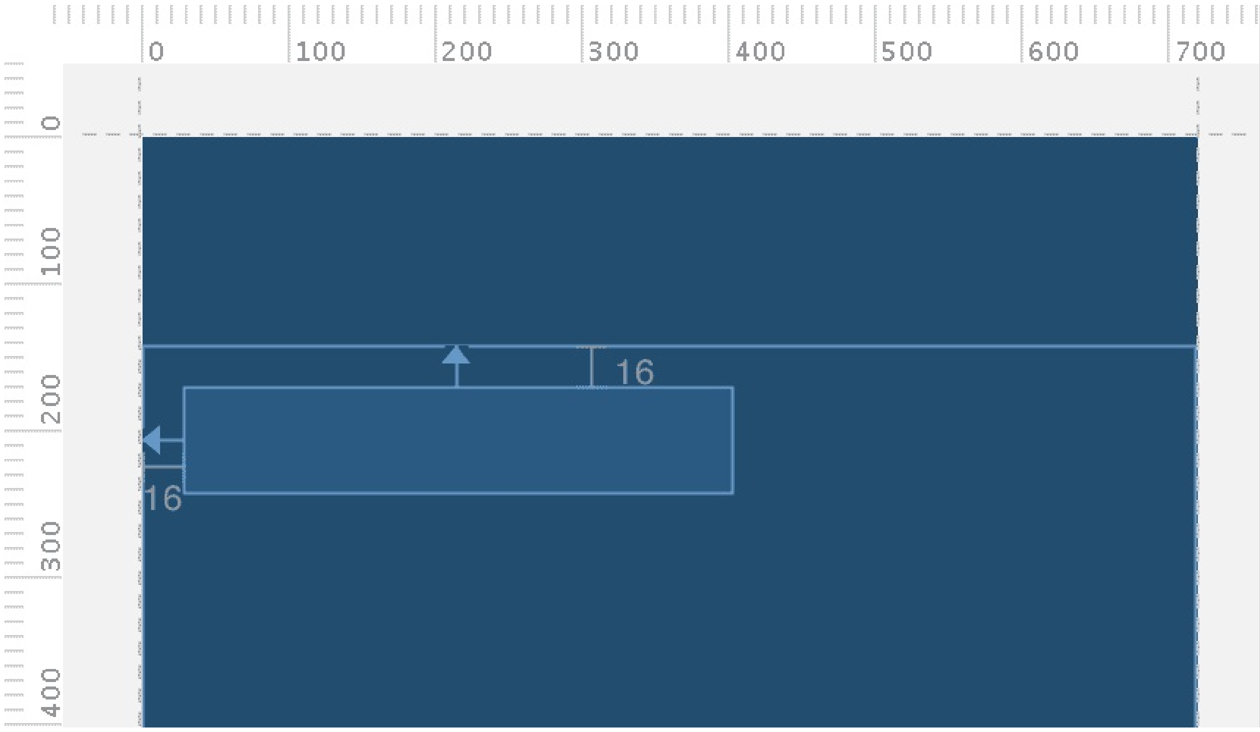


图 **4.** 文本框与父布局顶部和左边形成约束

添加一个按钮

1.同样的，在 **Palette** 面板左侧部分选中 **Widgets** 分类，然后拖出 **Button** 并放到编辑器中靠近父布局右上角的地方。

2.在 Button 的左侧与 EditText 右侧建立一个约束。

3.针对可显示文字的 View ，我们可以通过在每个 View 的文字基线之间建立约束从而使得它们水平对齐。在编辑器中选中一个 View ，这个被选中的 View 下方会出现一个 **Baseline Constraint** 

按钮。例如选中本例中的 Button ，Button 里面会出现一个线状的锚点，将这个锚点拖放到 EditText 中的基线锚点上。

现在可以看到的效果如图5所示。

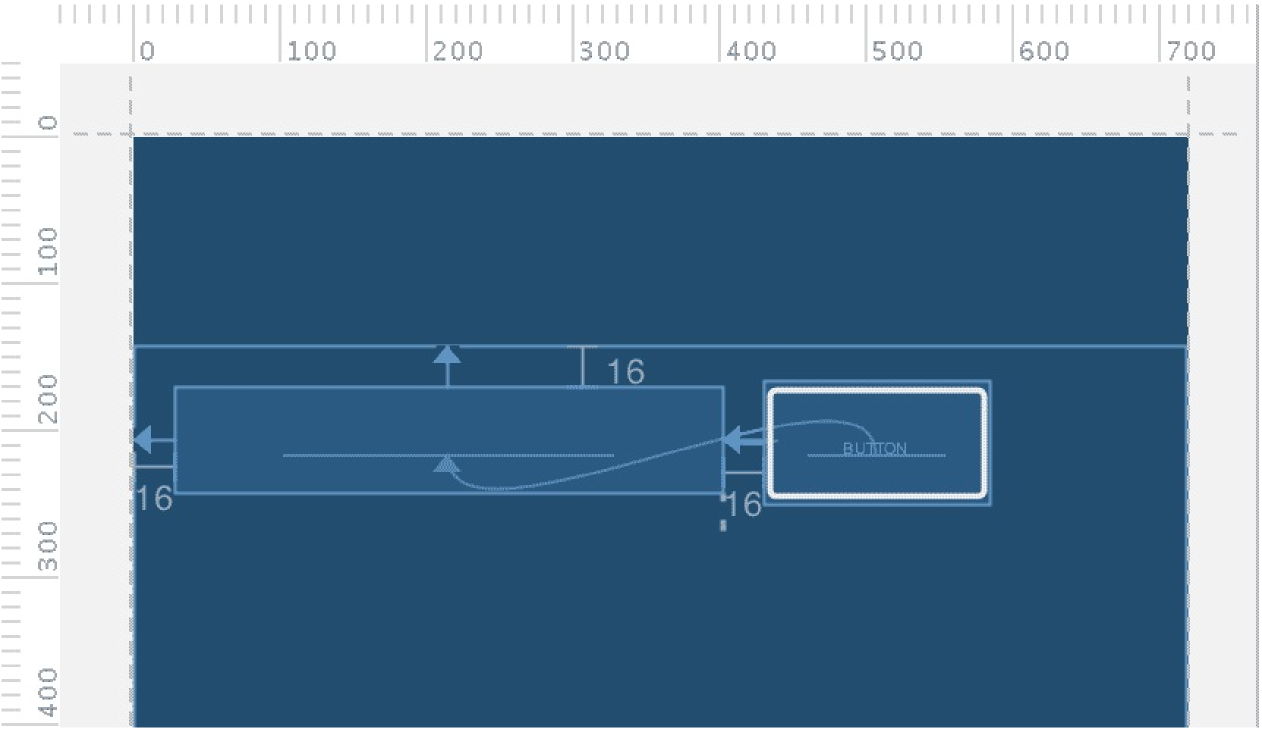


图 **5.** Button 左侧和 EditText 右侧以及彼此的基线之间建立了约束

> 注意： 我们也可以用 Button 顶部或底部的锚点建立约束从而达到水平对齐的目的，但是由于在 Button 内部是有一个 padding 值的，所以通过这种方式建立约束并不会眞正实现水平对齐。

改变 UI 中显示的字符串

点击工具栏中的 **Show Design** 按钮可以预览我们的 UI，可以看到 EditText 默认显示的字符串是 “Name”， Button 默认显示的字符串是 “Button”。接下来我们的目的就是修改这些字符串。

1.打开 **Project** 面板，然后打开文件 `app/res/values/strings.xml` 。

`strings.xml` 是一个字符串资源文件，我们应该把 UI 布局中出现的字符串定义在这个文件中。相比于在布局或逻辑代码中硬编码，这样在一个文件集中管理所有的字符串更利于字符串的查找、修改甚至是本地化操作。

2.点击右上角的 **Open editor** 按钮可以打开 **Translations Editor**，在这个编辑器中不仅可以增加、修改默认字符串，还能很好的管理所有字符串的翻译版本。

3.点击左上角 **Add Key** 按钮为 EditText 新增一个提示文字（hint text）：

1.在 Key 那一栏填入 "edit_message" , 这就是这个字符串的 id。

2.在 Default Value 那一栏填入 "Enter a message" ，这就是字符串的内容，会显示到 UI 中。

3.点击 **OK**。

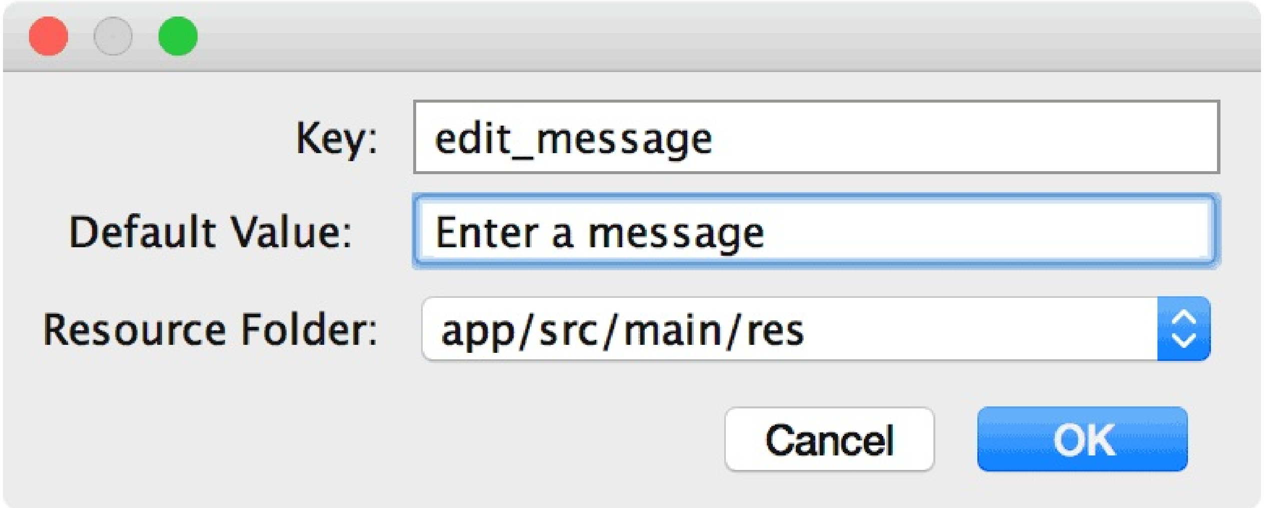


图 **6.** 新增字符串资源的对话框

4.新增另一个字符串资源， Key 为 "button_send" ，Default Value 为 "Send"。

现在可以通过点击标签栏的 **activity_main.xml** 返回布局文件通过以下步骤为每个 View 设置相应的字符串资源：

1.在布局编辑器中选中 EditText 对象，如果在窗口右侧没有出现 **Propreties** 面板的话可以点击右边侧边栏中的 **Properities** 按钮。 **Propreties** 面板会显示选中对象的属性。

2.在 **Propreties** 面板中找到 *hint* 属性，然后点击文本框右边的 **Pick a Resource** ... 按钮，在弹出的对话框中双击 **edit_message**。

3.同样在 EditText 的 **Propreties** 面板中删除 *text* 属性的值（当前值为 "Name"）。

4.在布局编辑器中选中 Button 对象切换到 Button 对应的 **Propreties** 面板，将 Button 的 *text* 属性值更换成 id 为 "button_send" 的字符串资源。

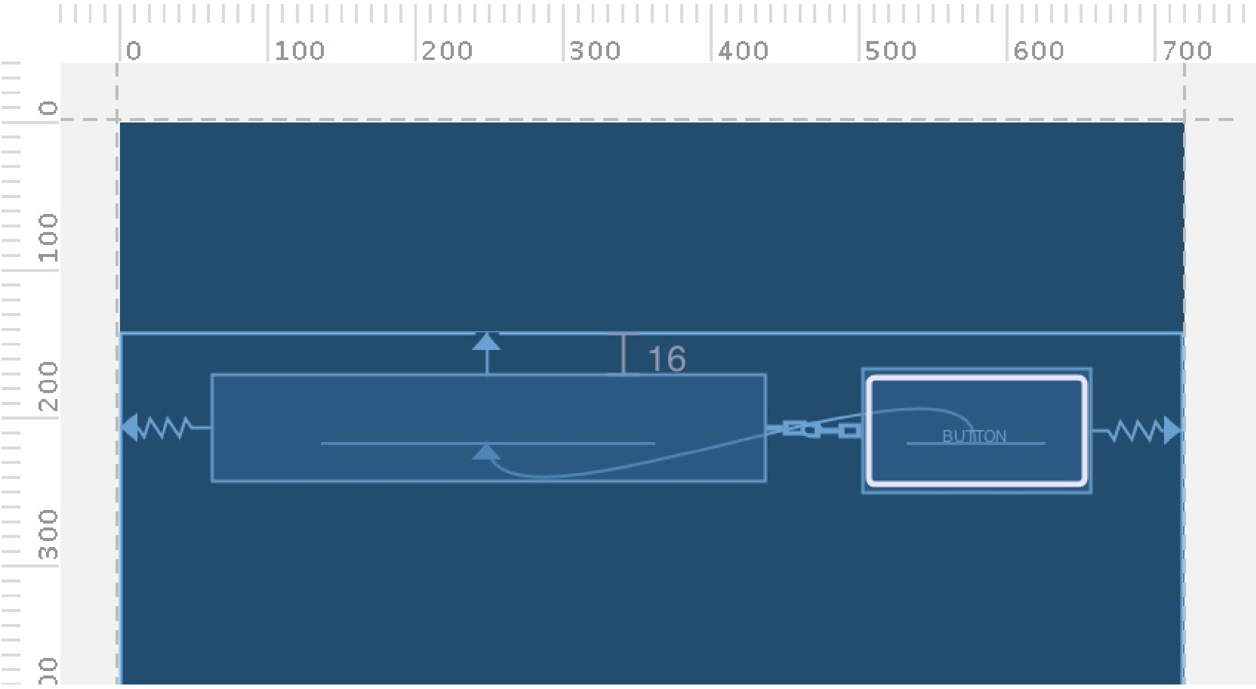
让文本输入框大小灵活

为了创建一个可以适应不同大小的屏幕，我们需要调整 EditText，使得它可以在计算完 Button 的宽度和 Margin 间距之后，自行伸展至占有所有的剩余宽度。

在继续之前，点击 **Show Blueprint** 按钮，我们依然在蓝图模式下工作。

1.选择所有的 View 对象（选中其中一个，按住 Shift 并选中另一个），鼠标右击其中一个 View 对象，从菜单中选择 **Center Horizontally**。

虽然我们的目标不是让所有的 View 对象水平居中，但是这种方法可以在这些 View 之间快速建立起一个约束链(constraint chain)。约束链是在两个或多个 View 对象之间形成的一个双向约束，它可以将这些 View 对象链接起来多为一个整体进行编排布局。不过这样会消除 View 对象之间水平方向的间距，所以后面需要手动更改。设置完约束链的效果如下图：



2.选中 Button 并打开相应的 **Propreties** 面板，将左右 margin 设置为16。

3.选中 EditText 并将 left margin 设置为16。

4.在 EditText 的 **Propreties** 面板中，点击图8中标示为1处的按钮（这是宽度指示符）直到出现  为止，这表示我们已经将 EditText 的 *width* 属性设置为 **Match Constraints** 了。

"Match Constraints"的意思是 View 的宽度受水平方向的约束和间距影响。因此，EditText 的宽度会伸展至占用所有剩余的水平空间（在计算完 Button 的宽度和 Margin 间距之后）。

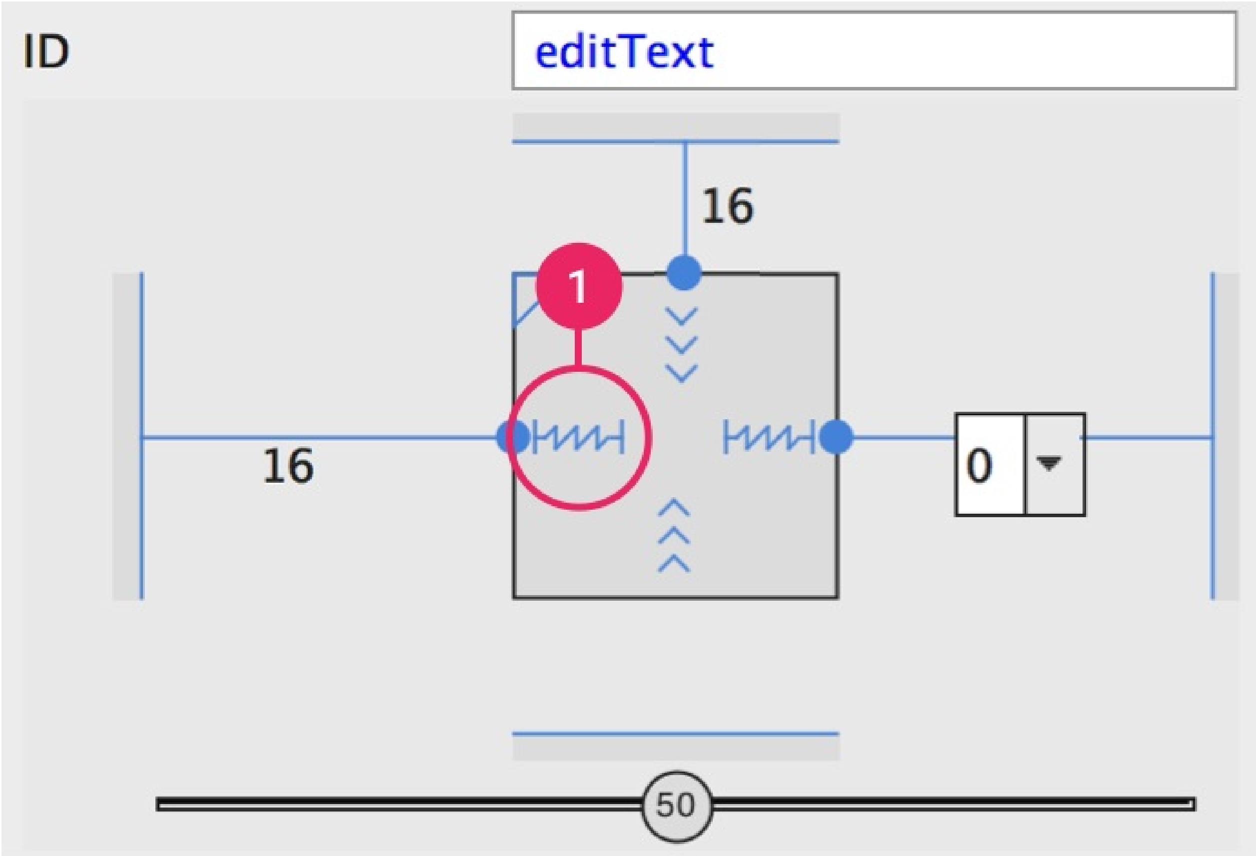


图 **8.** 设置 width 属性值为 "Match Constraints"

目前为止，我们已经完成了本节课程中布局的所有内容。最终效果应该如图9所示。

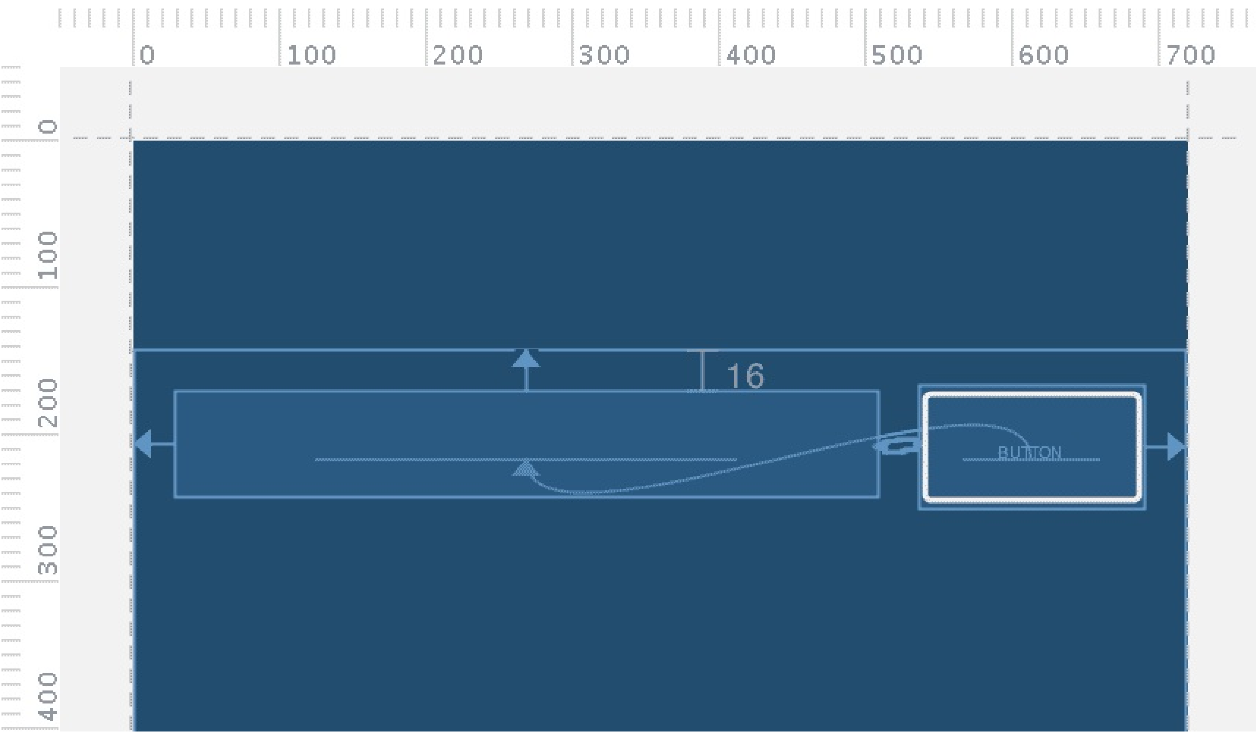


图 **9.** EditText 占有所有剩余空间

如果您的布局没有达到预期的效果，可以查看下面的完整代码进行对比（各属性出现的顺序不会影响布局的样式）。以下是完整代码：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.myfirstapp.MainActivity">

    <EditText
        android:id="@+id/editText"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:ems="10"
        android:hint="@string/edit_message"
        android:inputType="textPersonName"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintRight_toLeftOf="@+id/button"
        android:layout_marginLeft="16dp" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send"
        app:layout_constraintBaseline_toBaselineOf="@+id/editText"
        app:layout_constraintLeft_toRightOf="@+id/editText"
        app:layout_constraintRight_toRightOf="parent"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp" />
</android.support.constraint.ConstraintLayout>
```

想要了解更多关于 chain 的信息或者更多关于 *ConstraintLayout* 的使用方法，可以参考Build a Responsive UI with ConstraintLayout。

运行我们的 app

如果在上一课中已经将 app 安装在设备上了，只要点击工具栏中 **Apply Changes** 按钮就可以将最新的布局更新到手机上。或者点击 **Run** 按钮将 app 安装到手机上并运行。

目前为止，当我们点击 Button 时仍然不会有任何反应，下一课中我们将完善 app，点击 Button 时会启动另一个 Activity。

下一节：启动另一个 Activity

启动另一个Activity

> 编写:crazypudding - 原文:http://developer.android.com/training/basics/firstapp/starting-activity.html

在完成上一课(建立简单的用户界面)后，我们已经拥有了显示一个 activity（一个界面）的 app（应用），该 activity 包含了一个文本字段和一个按钮。在这节课中，你将添加一些新的代码到 `MyActivity` 中，当用户点击发送(Send)按钮时启动一个新的activity。

> 注意：本课程内容期待的运行环境为 Android Studio 2.3及以上

响应Send(发送)按钮

按以下步骤在 `MainActivity.java` 文件中新增一个方法，该方法会在我们点击 Send 按钮时触发：

1.打开文件 `app/java/com.example.myfirstapp/MainActivity.java` ，在其中添加一个 `sendMessage()` 方法存根（Method Stub）：

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** 当用户点击 Send 按钮时调用该方法 */
    public void sendMessage(View view) {
        // 此处的代码会在点击 Send 按钮时执行
    }
}
```

这里可能会出现名为 "Cannot resolve symbol" 的报错，在方法参数 View 下面会出现一条红色的波浪线，这是因为 Android Studio 不能解析 `View` 类。将光标移动到 View 上，然后按下 Alt + Enter （Mac中为 Option + Return）组合键快速修复。（如果出现菜单，则选择 Import class）

2.现在回到 `activity_main.xml` 文件，完成对 sendMessage() 方法的调用：

```
1.在布局编辑器中选中 Buton 对象

2.在 **Property** 面板中找到 *onClick* 属性，在下拉列表中选中 **sendMessage [MainActivity]**
```

完成这些操作后，当点击 Send 按钮时，系统会调用 sendMessage() 方法。

为保证系统能将 sendMessage() 方法与 android:onclick 成功匹配，这个方法需要满足以下要求：

- 方法的访问修饰符为 public
- 无返回值
- 只有一个 View 类型的参数（代表被点击的 View 对象）

接下来，你可以在这个方法中编写读取文本内容，并将该内容传到另一个Activity的代码。

构建一个Intent

Intent 是一个可以为不同组件在运行时提供链接的对象，例如为两个 Activity 提供链接。Intent 代表一个 app "想要做某事的意向"，你可以使用它来完成各种各样的任务，不过在本节课程中，我们只使用 intent 来启动另一个 Activity。

在 `MainActivity.java` 文件中，添加一个 EXTRA_MESSAGE 常量并完善 sendMesage() 方法中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {
    public static final String EXTRA_MESSAGE = "com.example.myfirstapp.MESSAGE";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** 当用户点击 Send 按钮时调用该方法 */
    public void sendMessage(View view) {
        Intent intent = new Intent(this, DisplayMessageActivity.class);
        EditText editText = (EditText) findViewById(R.id.editText);
        String message = editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
}
```

Android Studio 可能会再次出现 "Cannot resolve symbol" 的错误，同样使用 Alt + Enter（Mac中为 Option + Return）组合键快速导入类，完成后该类的导入项如下所示：

```
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
```

不过对 `DisplayMessageActivity` 的引用仍然会报错，因为这个类还不存在；暂时先忽略这个错误，我们很快就会解决这个问题。

以下是 sendMessage() 方法中要注意的几个地方：

1. Intent 构造方法中有两个参数：
2. 第一个参数是 Context (之所以用 `this` 是因为 Activity 类是 `Context` 的子类)
3. 接受系统发送 Intent 的应用组件对应的 Class（在这个案例中，指将要被启动的activity）
4. putExtra() 方法将从 EditText 中取到的值附加到 Intent 上。 Intent 可以以键-值对的方式携带数据，这些数据称为 *extras*。此处的键是一个 public 修饰的常量——EXTRA_MESSAGE，因为在另一个 Activity 中，我们需要以这个键来获取它对应的值。以应用包名为前缀来定义 intent extras 的键是一个很好的习惯，这使得 app 在与其他 app 交互的过程中能保证这个键的唯一性。
5. startActivity() 方法启动了 Intent 定义的 `DisplayMessageActivity` 的实例。现在我们需要新建一个 `DisplayMessageActivity` 类。

创建第二个Activity

1.在 **Project** 面板中，右击 **app** 文件夹，依次选择 `New > Activity > Empty Activity` 。

2.在弹出的 **Configure Activity** 面板中，将 *Activity Name* 的值修改为 "DispalyMessageActivity" ，其他属性保持默认然后点击 **finsh**。

在这个过程中，Android Studio 自动完成了一下三件事：

- 创建了一个名为 `DisplayMessageActivity.java` 的文件。
- 创建一个相应的布局文件 `activity_display_message.xml` 。
- 在 `AndroidManifest.xml` 文件中为该文件添加了对应的 \ 标签（没有这个标签将不能启动相应的 Activity）。

如果现在运行 app 并点击第一个 Activity 中的 Send 按钮，app 会跳转到第二个 Activity（也就是刚新建的 DisplayMessageActivity）但是显示一片空白。这是因为新建的 Activity 默认使用模板提供的空白布局页（activity_display_message）。

新增一个 TextView

由于新建的 Activity 引用了一个空白的布局页，所以我们现在在这个布局页中添加一个 TextView 用来显示信息。

1.打开文件 `app/res/layout/activity_display_message.xml` 。

2.打开自动连接功能，点击工具栏中的 **Turn On Autoconnect**  按钮。

3.在 **Pallete** 面板中选中 TextView ，将它拖到布局中靠近父布局顶部并且大约水平居中的位置，当在布局中央会出现一条虚线时放下。这步操作后，ConstraintLayout 的自动连接功能(Autoconnect)为 TextView 新增了相应的约束使其水平居中。

4.为 TextView 的顶部和父布局顶部新增一个约束，这时效果图如图1。

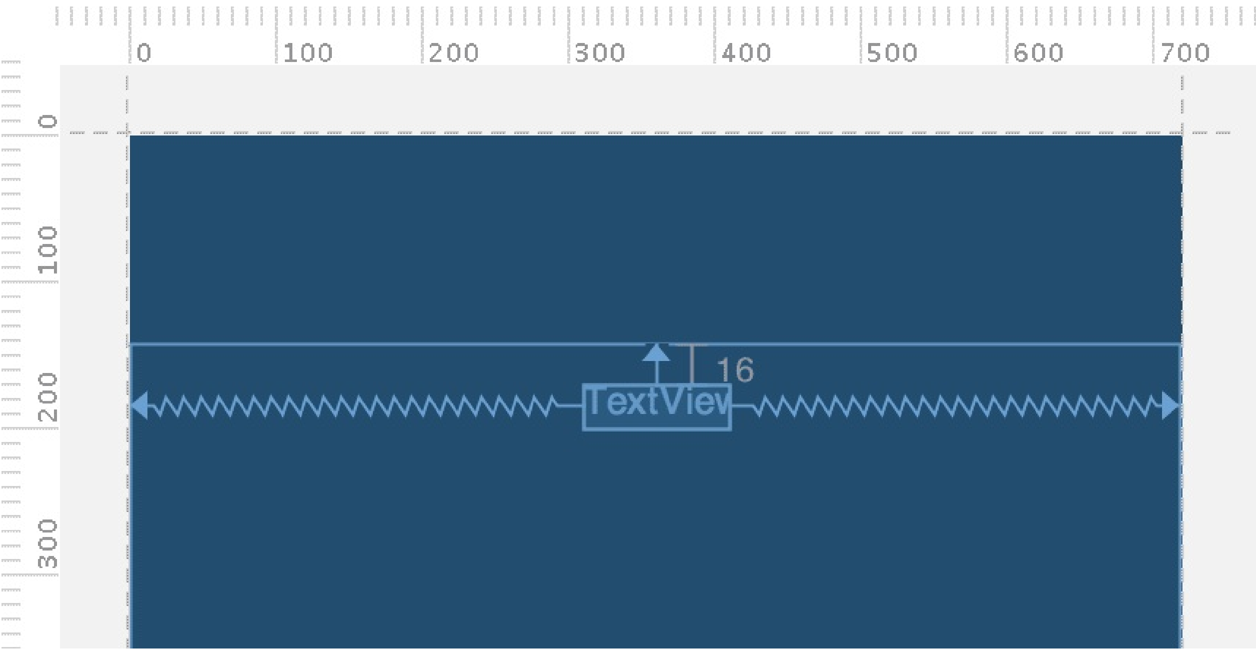


图 **1.** TextView 在布局中水平居中

当然，也可以为 TextView 做一些样式调整。在 **Properties** 面板中展开 **TextAppearance** 选项改变其中一些属性的值，比如 *textSize* 和 *textColor*。

显示消息

现在我们来修改第二个 Activity，修改完成便可以接收第一个 Activity 发来的消息。

1.在 `DisplayMessageActivity.java` 文件中，往 `onCreate()` 方法添加一下代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_display_message);

    // 获取启动此 Activity 的 Intent 并从中取得附带的消息
    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);

    // 获取布局中 TextView 并为其设置文本信息
    TextView textView = (TextView) findViewById(R.id.textView);
    textView.setText(message);
}
```

2.利用组合键 Alt + Enter（Mac中为 Option + Return）导入需要的类。完成后该类的导入项如下：

```
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.TextView;
```

添加向上导航（Up Navigation）

我们应该为 app 中所有不是主要入口的页面添加导航，这样一来用户便可以通过 app bar 中的 Up 按钮返回到当前页面的逻辑父页面。

我们所需要做的就是在 [AndroidManifest.xml] 文件中为声明哪一个 Activity 是它的逻辑父项。打开清单文件，`app/Manifest/AndroidManifest.xml` ,在名为 DisplayMessageActivity 的标签中新增一下内容：

```
<activity android:name=".DisplayMessageActivity"
          android:parentActivityName=".MainActivity" >
    <!-- meta-data 标签是为了兼容 API 15 及以下的设备 -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>
```

现在 Android 系统已经自动在 DisplayMessageActivity 的 app bar 中添加了 Up 按钮。

运行 app

现在点击工具栏中的 **Apply Changes**  按钮再次运行 app。运行成功之后，试着在 EditText 中输入文字信息如："Hello world!"并点击 Send 按钮，你会看到信息已经显示在第二个 Activity 中了。如图：



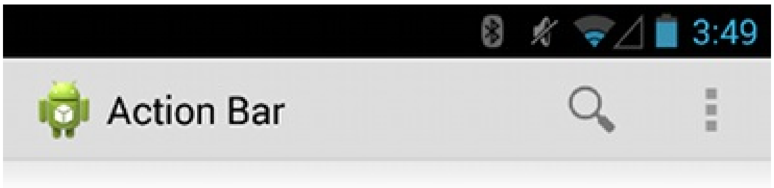
到此为止，已经创建好我们的第一个Android应用了！想要继续学习 Android 应用开发的基础知识，通过下面的链接进入到下一课吧。

添加Action Bar

> 编写:Vincent 4J - 原文:http://developer.android.com/training/basics/actionbar/index.html

Action Bar是我们可以为activity实现的最重要的设计元素之一。其提供了多种 UI 特性，可以让我们的 app 与其他 Android app 保持较高的一致性，从而为用户所熟悉。核心的功能包括：

- 一个专门的空间用来显示你的app的标识，以及指出目前所处在app的哪个页面。
- 以一种可预见的方式访问重要的操作（比如搜索）。
- 支持导航和视图切换（通过Tabs和下拉列表）



本章为 action bar 的基本知识提供了一个快速指南。关于 action bar 的更多特性，请查看 Action Bar 指南。

Lessons

- **建立ActionBar**

  学习如何为 activity 添加一个基本的 action bar，是仅仅支持 Android 3.0及以上的版本，还是同时也支持至Android 2.1的版本（通过使用 Andriod Support Library）。

- **添加Action按钮**

  学习如何在 action bar 中添加和响应用户操作。

- **ActionBar的风格化**

  学习如何自定义 action bar 的外观。

- **ActionBar覆盖叠加**

  学习如何在布局上面叠加 action bar，允许 action bar 隐藏时无缝过渡。

建立ActionBar

> 编写:Vincent 4J - 原文:http://developer.android.com/training/basics/actionbar/setting-up.html

Action bar 最基本的形式，就是为 Activity 显示标题，并且在标题左边显示一个 app icon。即使在这样简单的形式下，action bar对于所有的 activity 来说是十分有用的。它告知用户他们当前所处的位置，并为你的 app 维护了持续的同一标识。

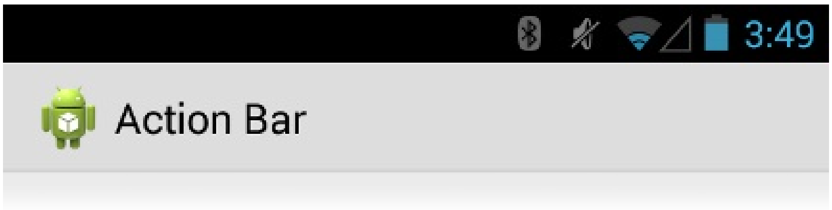


图 1. 一个有 app icon 和 Activity 标题的 action bar

设置一个基本的 action bar，需要 app 使用一个 activity 主题，该主题必须是 action bar 可用的。如何声明这样的主题取决于我们 app 支持的 Android 最低版本。本课程根据我们 app 支持的 Android 最低版本分为两部分。

仅支持 Android 3.0 及以上版本

从 Android 3.0(API lever 11) 开始，所有使用 Theme.Holo 主题（或者它的子类）的 Activity 都包含了 action bar，当 targetSdkVersion 或 minSdkVersion 属性被设置成 “11” 或更大时，它是默认主题。

所以，要为 activity 添加 action bar，只需简单地设置属性为 `11` 或者更大。例如：

```
<manifest ... >
    <uses-sdk android:minSdkVersion="11" ... />
    ...
</manifest>
```

> 注意: 如果创建了一个自定义主题，需确保这个主题使用一个 Theme.Holo的主题作为父类。详情见 Action bar 的风格化

到此，我们的 app 使用了 `Theme.Holo` 主题，并且所有的 activity 都显示 action bar。

支持 Android 2.1 及以上版本

当 app 运行在 Andriod 3.0 以下版本（不低于 Android 2.1）时，如果要添加 action bar，需要加载 Android Support 库。

开始之前，通过阅读Support Library Setup文档来建立**v7 appcompat** library（下载完library包之后，按照Adding libraries with resources的指引进行操作）。

在 Support Library集成到你的 app 工程中之后：

1、更新 activity，以便于它继承于 ActionBarActivity。例如：

```
public class MainActivity extends ActionBarActivity { ... }
```

2、在 mainfest 文件中，更新 `<application>` 标签或者单一的 `<activity>` 标签来使用一个 Theme.AppCompat 主题。例如：

```
<activity android:theme="@style/Theme.AppCompat.Light" ... >
```

> 注意: 如果创建一个自定义主题，需确保其使用一个 `Theme.AppCompat` 主题作为父类。详情见 Action bar 风格化

现在，当 app 运行在 Android 2.1(API level 7) 或者以上时，activity 将包含 action bar。

切记，在 manifest 中正确地设置 app 支持的 API level：

```
<manifest ... >
    <uses-sdk android:minSdkVersion="7"  android:targetSdkVersion="18" />
    ...
</manifest>
```

添加Action按钮

> 编写:[Vincent 4J](#) - 原文:http://developer.android.com/training/basics/actionbar/adding-buttons.html

Action bar 允许我们为当前环境下最重要的操作添加按钮。那些直接出现在 action bar 中的 icon 和/或文本被称作**action buttons(操作按钮)**。安排不下的或不足够重要的操作被隐藏在 **action overflow** （超出空间的action，译者注）中。

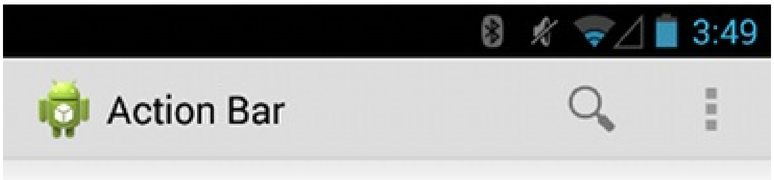


图 1. 一个有search操作按钮和 action overflow 的 action bar，在 action overflow 里能展现额外的操作。

在 XML 中指定操作

所有的操作按钮和 action overflow 中其他可用的条目都被定义在 menu资源 的 XML 文件中。通过在项目的 `res/menu` 目录中新增一个 XML 文件来为 action bar 添加操作。

为想要添加到 action bar 中的每个条目添加一个 `<item>` 元素。例如：

`res/menu/main_activity_actions.xml`

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- 搜索, 应该作为动作按钮展示-->
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search"
          android:title="@string/action_search"
          android:showAsAction="ifRoom" />
    <!-- 设置, 在溢出菜单中展示 -->
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:showAsAction="never" />
</menu>
```

上述代码声明，当 action bar 有可用空间时，搜索操作将作为一个操作按钮来显示，但设置操作将一直只在 action overflow 中显示。（默认情况下，所有的操作都显示在 action overflow 中，但为每一个操作指明设计意图是很好的做法。）

icon 属性要求每张图片提供一个 `resource ID` 。在 `@drawable/` 之后的名字必须是存储在项目目录 `res/drawable/` 下位图图片的文件名。例如： `ic_action_search.png` 对应 "@drawable/ic_action_search"。同样地，title 属性使用通过 XML 文件定义在项目目录 `res/values/` 中的一个 `string 资源` ，详情请参见 创建一个简单的 UI 。

> 注意：当创建 icon 和其他 bitmap 图片时，要为不同屏幕密度下的显示效果提供多个优化的版本，这一点很重要。在 支持不同屏幕 课程中将会更详细地讨论。

如果为了兼容 **Android 2.1** 的版本使用了 **Support** 库，在 `android` 命名空间下 `showAsAction` 属性是不可用的。Support 库会提供替代它的属性，我们必须声明自己的 XML 命名空间，并且使用该命名空间作为属性前缀。（一个自定义 XML 命名空间需要以我们的 app 名称为基础，但是可以取任何想要的名称，它的作用域仅仅在我们声明的文件之内。）例如：

`res/menu/main_activity_actions.xml`

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <!-- 搜索, 应该展示为动作按钮 -->
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search"
          android:title="@string/action_search"
          yourapp:showAsAction="ifRoom"  />
    ...
</menu>
```

为 Action Bar 添加操作

要为 action bar 布局菜单条目，就要在 activity 中实现 onCreateOptionsMenu() 回调方法来 `inflate` 菜单资源从而获取 Menu 对象。例如：

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // 为ActionBar扩展菜单项
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

为操作按钮添加响应事件

当用户按下某一个操作按钮或者 action overflow 中的其他条目，系统将调用 activity 中 onOptionsItemSelected()的回调方法。在该方法的实现里面调用MenuItem的getItemId()来判断哪个条目被按下 - 返回的 ID 会匹配我们声明对应的 `<item>` 元素中 `android:id` 属性的值。

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // 处理动作按钮的点击事件
    switch (item.getItemId()) {
        case R.id.action_search:
            openSearch();
            return true;
        case R.id.action_settings:
            openSettings();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

为下级 Activity 添加向上按钮

在不是程序入口的其他所有屏中（activity 不位于主屏时），需要在 action bar 中为用户提供一个导航到逻辑父屏的**up button(**向上按钮**)**。

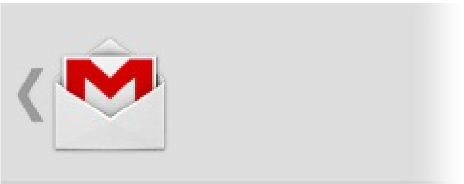


图 2. Gmail 中的 up button。

当运行在 Android 4.1(API level 16) 或更高版本，或者使用 Support 库中的 ActionBarActivity 时，实现向上导航需要在 manifest 文件中声明父 activity ，同时在 action bar 中设置向上按钮可用。

如何在 manifest 中声明一个 activity 的父类，例如：

```
<application ... >
    ...
    <!-- 主 main/home 活动 (没有上级活动) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- 主活动的一个子活动-->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!--  meta-data 用于支持 support 4.0 以及以下来指明上级活动 -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

然后，通过调用setDisplayHomeAsUpEnabled() 来把 app icon 设置成可用的向上按钮：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_displaymessage);

    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    // 如果你的minSdkVersion属性是11或更高, 应该这么用:
    // getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

由于系统已经知道 `MainActivity` 是 `DisplayMessageActivity` 的父 activity，当用户按下向上按钮时，系统会导航到恰当的父 activity - 你不需要去处理向上按钮的事件。

更多关于向上导航的信息，请见 提供向上导航。

自定义ActionBar的风格

> 编写:Vincent 4J - 原文:http://developer.android.com/training/basics/actionbar/styling.html

Action bar 为用户提供一种熟悉可预测的方式来展示操作和导航，但是这并不意味着我们的 app 要看起来和其他 app 一样。如果想将 action bar 的风格设计的合乎我们产品的定位，只需简单地使用 Android 的 样式和主题 资源。

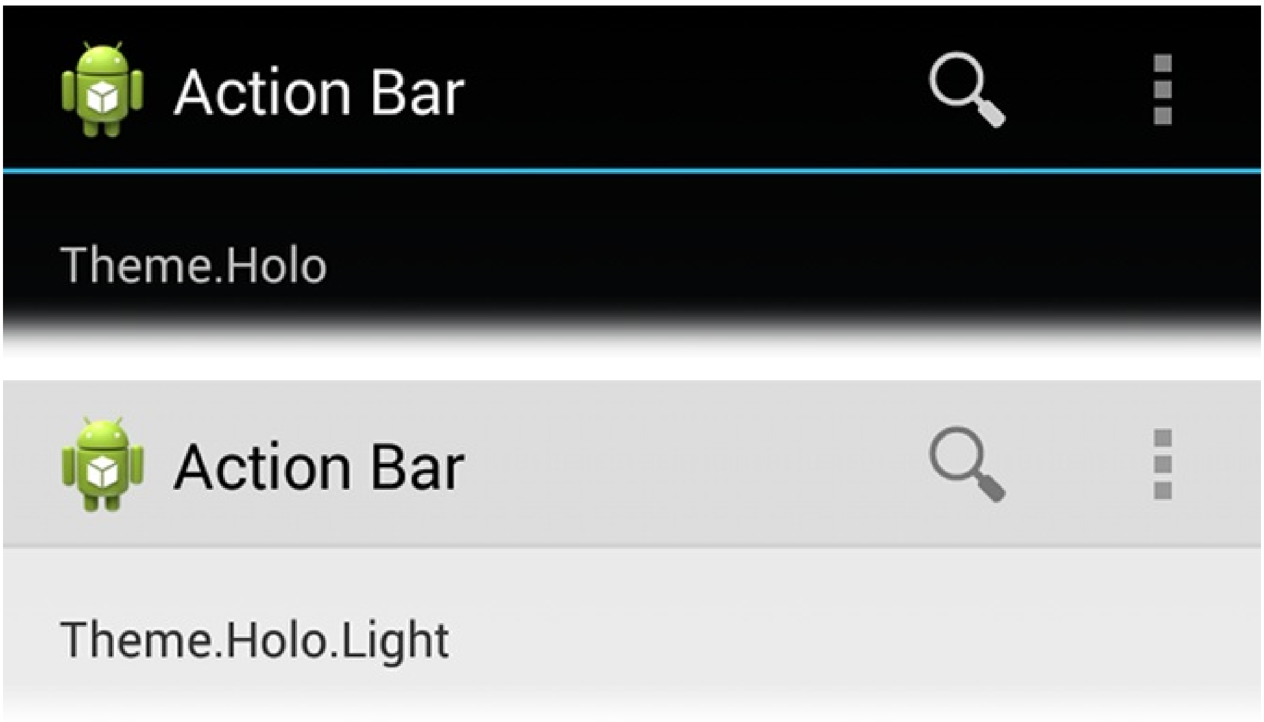
Android 包括一少部分内置的 activity 主题，这些主题中包含 “dark” 或 “light” 的 action bar 样式。我们也可以扩展这些主题，以便于更好的为 action bar 自定义外观。

> 注意：如果我们为 action bar 使用了 Support 库的 API，那我们必须使用（或重写） Theme.AppCompat 家族样式（甚至 Theme.Holo 家族，在 API level 11 或更高版本中可用）。如此一来，声明的每一个样式属性都必须被声明两次：一次使用系统平台的样式属性（android: 属性），另一次使用 Support 库中的样式属性（appcompat.R.attr 属性 - 这些属性的上下文其实就是我们的 app）。更多细节请查看下面的示例。

使用一个 Android 主题

Android 包含两个基本的 activity 主题，这两个主题决定了 action bar 的颜色：

- Theme.Holo，一个 “dark” 的主题
- Theme.Holo.Light，一个 “light” 的主题

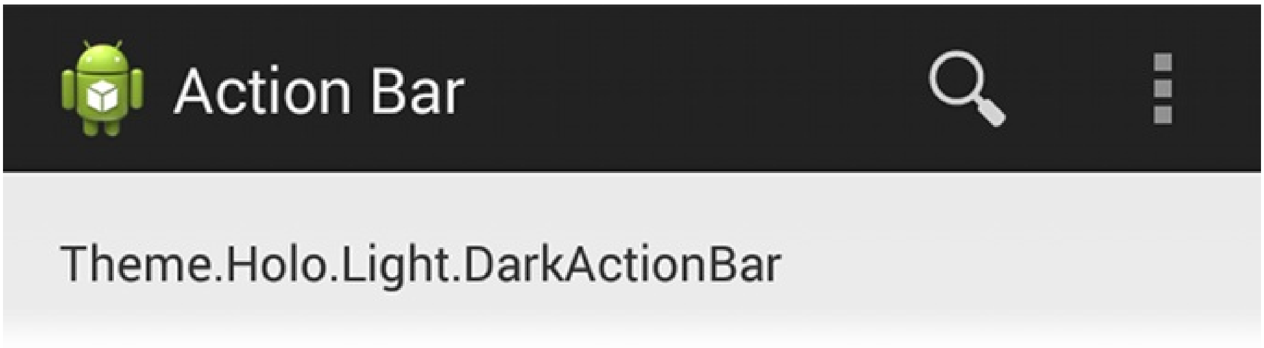


这些主题即可以被应用到 app 全局，也可以通过在 manifest 文件中设置 `<application>` 元素或 `<activity>` 元素的 `android:theme` 属性，对单一的 activity 进行设置。

例如：

```
<application android:theme="@android:style/Theme.Holo.Light" ... />
```

可以通过声明 activity 的主题为 Theme.Holo.Light.DarkActionBar 来达到如下效果：action bar 为dark，其他部分为light。



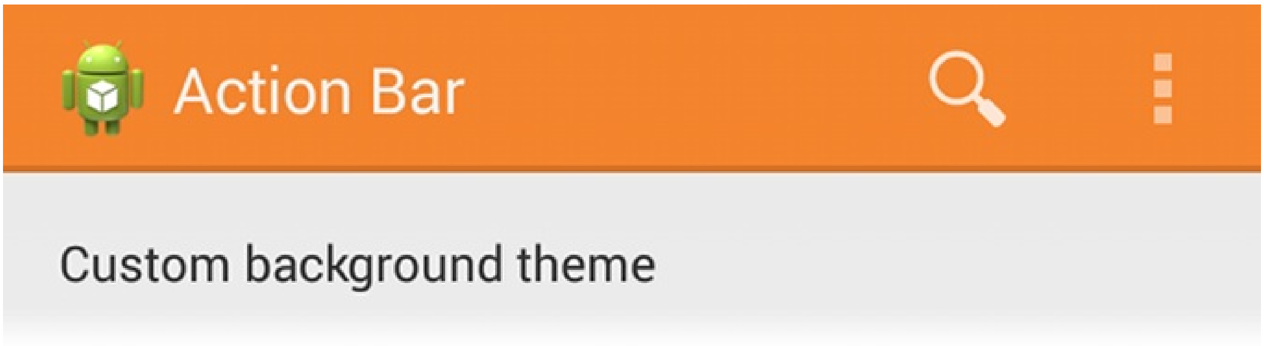
当使用 Support 库时，必须使用 Theme.AppCompat 主题替代：

- Theme.AppCompat，一个“dark”的主题
- Theme.AppCompat.Light，一个“light”的主题
- Theme.AppCompat.Light.DarkActionBar，一个带有“dark” action bar 的“light”主题

一定要确保我们使用的 action bar icon 的颜色与 action bar 本身的颜色有差异。Action Bar Icon Pack 为 Holo “dark”和“light”的 action bar 提供了标准的 action icon。

自定义背景

为改变 action bar的背景，可以通过为 activity 创建一个自定义主题，并重写 actionBarStyle 属性来实现。actionBarStyle 属性指向另一个样式；在该样式里，通过指定一个 drawable 资源来重写 background 属性。



如果我们的 app 使用了 navigation tabs 或 split action bar ，也可以通过分别设置 backgroundStacked 和 backgroundSplit 属性来为这些条指定背景。

> **Note**：为自定义主题和样式声明一个合适的父主题，这点很重要。如果没有父样式，action bar将会失去很多默认的样式属性，除非我们自己显式的对他们进行声明。

仅支持 Android 3.0 和更高

当仅支持 Android 3.0 和更高版本时，可以通过如下方式定义 action bar 的背景：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或者活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.Holo.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar 样式 -->
    <style name="MyActionBar"
           parent="@android:style/Widget.Holo.Light.ActionBar.Solid.Inverse">
        <item name="android:background">@drawable/actionbar_background</item>
    </style>
</resources>
```

然后，将主题应用到 app 全局或单个的 activity 之中：

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

支持 Android 2.1 和更高

当使用 Support 库时，上面同样的主题必须被替代成如下：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或者活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>

        <!-- 支持库兼容 -->
        <item name="actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar 样式 -->
    <style name="MyActionBar"
           parent="@style/Widget.AppCompat.Light.ActionBar.Solid.Inverse">
        <item name="android:background">@drawable/actionbar_background</item>

        <!-- 支持库兼容 -->
        <item name="background">@drawable/actionbar_background</item>
    </style>
</resources>
```

然后，将主题应用到 app 全局或单个的 activity 之中：

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

自定义文本颜色

修改 action bar 中的文本颜色，需要分别设置每个元素的属性：

- Action bar 的标题：创建一种自定义样式，并指定 `textColor` 属性；同时，在自定义的 actionBarStyle 中为 titleTextStyle 属性指定为刚才的自定义样式。

> 注意：被应用到 titleTextStyle 的自定义样式应该使用 TextAppearance.Holo.Widget.ActionBar.Title 作为父样式。

- Action bar tabs：在 activity 主题中重写 actionBarTabTextStyle
- Action 按钮：在 activity 主题中重写 actionMenuTextColor

仅支持 Android 3.0 和更高

当仅支持 Android 3.0 和更高时，样式 XML 文件应该是这样的：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或者活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.Holo">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="android:actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar 样式 -->
    <style name="MyActionBar"
           parent="@style/Widget.Holo.ActionBar">
        <item name="android:titleTextStyle">@style/MyActionBarTitleText</item>
    </style>

    <!-- ActionBar 标题文本 -->
    <style name="MyActionBarTitleText"
           parent="@style/TextAppearance.Holo.Widget.ActionBar.Title">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar Tab标签 文本样式 -->
    <style name="MyActionBarTabText"
           parent="@style/Widget.Holo.ActionBar.TabText">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>
</resources>
```

支持 Android 2.1 和更高

当使用 Support 库时，样式 XML 文件应该是这样的：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或者活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="android:actionMenuTextColor">@color/actionbar_text</item>

        <!-- 支持库兼容 -->
        <item name="actionBarStyle">@style/MyActionBar</item>
        <item name="actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar 样式 -->
    <style name="MyActionBar"
           parent="@style/Widget.AppCompat.ActionBar">
        <item name="android:titleTextStyle">@style/MyActionBarTitleText</item>

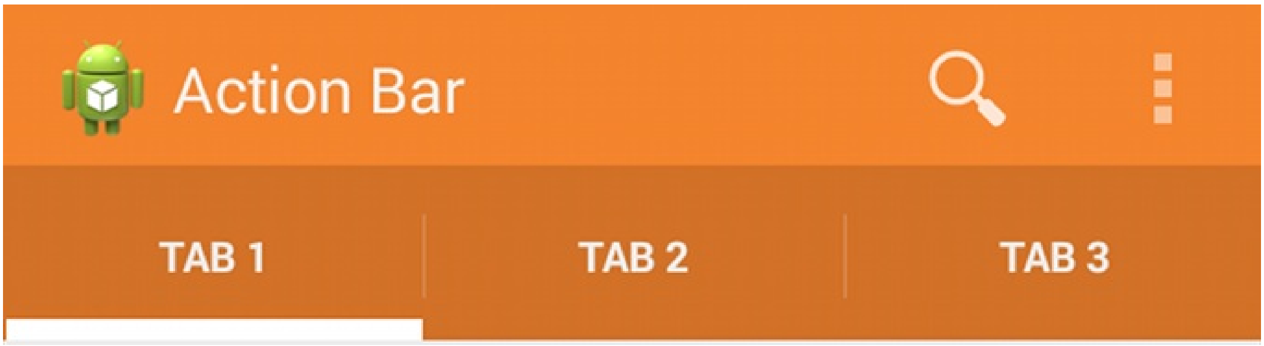
        <!-- 支持库兼容 -->
        <item name="titleTextStyle">@style/MyActionBarTitleText</item>
    </style>

    <!-- ActionBar 标题文本 -->
    <style name="MyActionBarTitleText"
           parent="@style/TextAppearance.AppCompat.Widget.ActionBar.Title">
        <item name="android:textColor">@color/actionbar_text</item>
        <!-- 文本颜色属性textColor是可以配合支持库向后兼容的 -->
    </style>

    <!-- ActionBar Tab标签文本样式 -->
    <style name="MyActionBarTabText"
           parent="@style/Widget.AppCompat.ActionBar.TabText">
        <item name="android:textColor">@color/actionbar_text</item>
        <!-- 文本颜色属性textColor是可以配合支持库向后兼容的 -->
    </style>
</resources>
```

自定义 Tab Indicator

为 activity 创建一个自定义主题，通过重写 actionBarTabStyle 属性来改变 navigation tabs 使用的指示器。actionBarTabStyle 属性指向另一个样式资源；在该样式资源里，通过指定一个 state-list drawable 来重写 background 属性。



> 注意：一个state-list drawable 是重要的，它可以通过不同的背景来指出当前选择的 tab 与其他 tab 的区别。更多关于如何创建一个 drawable 资源来处理多个按钮状态，请阅读 State List 文档。

例如，这是一个状态列表 drawable，为一个 action bar tab 的多种不同状态分别指定背景图片：

`res/drawable/actionbar_tab_indicator.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

<!-- 按钮没有按下的状态 -->

    <!-- 没有焦点的状态 -->
    <item android:state_focused="false" android:state_selected="false"
          android:state_pressed="false"
          android:drawable="@drawable/tab_unselected" />
    <item android:state_focused="false" android:state_selected="true"
          android:state_pressed="false"
          android:drawable="@drawable/tab_selected" />

    <!-- 有焦点的状态 (例如D-Pad控制或者鼠标经过) -->
    <item android:state_focused="true" android:state_selected="false"
          android:state_pressed="false"
          android:drawable="@drawable/tab_unselected_focused" />
    <item android:state_focused="true" android:state_selected="true"
          android:state_pressed="false"
          android:drawable="@drawable/tab_selected_focused" />


<!--  按钮按下的状态D -->

    <!-- 没有焦点的状态 -->
    <item android:state_focused="false" android:state_selected="false"
          android:state_pressed="true"
          android:drawable="@drawable/tab_unselected_pressed" />
    <item android:state_focused="false" android:state_selected="true"
        android:state_pressed="true"
        android:drawable="@drawable/tab_selected_pressed" />

    <!--有焦点的状态 (例如D-Pad控制或者鼠标经过)-->
    <item android:state_focused="true" android:state_selected="false"
          android:state_pressed="true"
          android:drawable="@drawable/tab_unselected_pressed" />
    <item android:state_focused="true" android:state_selected="true"
          android:state_pressed="true"
          android:drawable="@drawable/tab_selected_pressed" />
</selector>
```

仅支持 Android 3.0 和更高

当仅支持 Android 3.0 和更高时，样式 XML 文件应该是这样的：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.Holo">
        <item name="android:actionBarTabStyle">@style/MyActionBarTabs</item>
    </style>

    <!-- ActionBar tabs 标签样式 -->
    <style name="MyActionBarTabs"
           parent="@style/Widget.Holo.ActionBar.TabView">
        <!-- 标签指示器 -->
        <item name="android:background">@drawable/actionbar_tab_indicator</item>
    </style>
</resources>
```

支持 Android 2.1 和更高

当使用 Support 库时，样式 XML 文件应该是这样的：

`res/values/themes.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 应用于程序或活动的主题 -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat">
        <item name="android:actionBarTabStyle">@style/MyActionBarTabs</item>

        <!-- 支持库兼容 -->
        <item name="actionBarTabStyle">@style/MyActionBarTabs</item>
    </style>

    <!-- ActionBar tabs 样式 -->
    <style name="MyActionBarTabs"
           parent="@style/Widget.AppCompat.ActionBar.TabView">
        <!-- 标签指示器 -->
        <item name="android:background">@drawable/actionbar_tab_indicator</item>

        <!-- 支持库兼容 -->
        <item name="background">@drawable/actionbar_tab_indicator</item>
    </style>
</resources>
```

> 更多资源
>
> - 关于 action bar 的更多样式属性，请查看 Action Bar 指南
> - 学习更多样式的工作机制，请查看 样式和主题 指南
> - 全面的 action bar 样式，请尝试 Android Action Bar 样式生成器

ActionBar的覆盖叠加

> 编写:Vincent 4J - 原文:http://developer.android.com/training/basics/actionbar/overlaying.html

默认情况下，action bar 显示在 activity 窗口的顶部，会稍微地减少其他布局的有效空间。如果在用户交互过程中要隐藏和显示 action bar，可以通过调用 ActionBar 中的 hide()和show() 来实现。但是，这将导致 activity 基于新尺寸重新计算与绘制布局。

为避免在 action bar 隐藏和显示过程中调整布局的大小，可以为 action bar 启用叠加模式 (**overlay mode**)。在叠加模式下，所有可用的空间都会被用来布局就像ActionBar不存在一样，并且 action bar 会叠加在布局之上。这样布局顶部就会有点被遮挡，但当 action bar 隐藏或显示时，系统不再需要调整布局而是无缝过渡。

> **Note**：如果希望 action bar 下面的布局部分可见，可以创建一个背景部分透明的自定义式样的 action bar，如图 1 所示。关于如何定义 action bar 的背景，请查看 自定义ActionBar的风格。

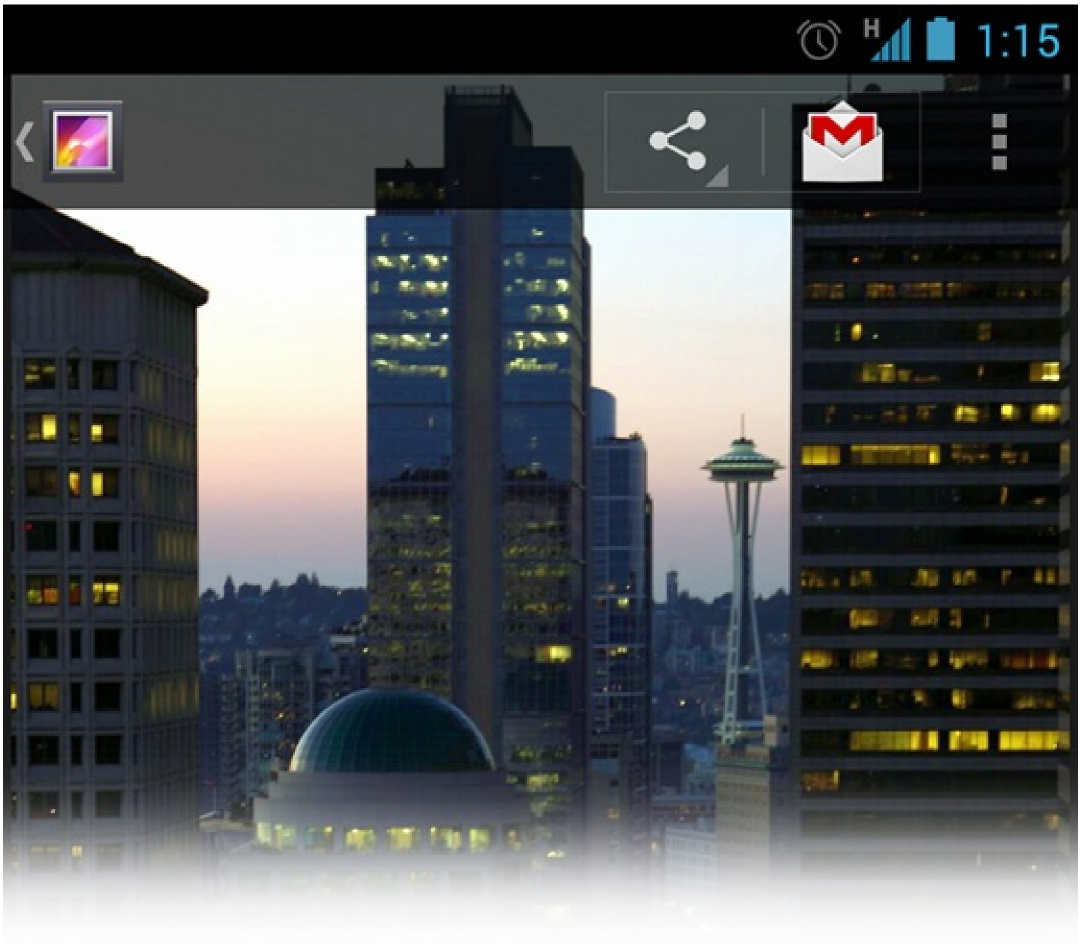


图 1. 叠加模式下的 gallery action bar

启用叠加模式(Overlay Mode)

要为 action bar 启用叠加模式，需要自定义一个主题，该主题继承于已经存在的 action bar 主题，并设置 `android:windowActionBarOverlay` 属性的值为 `true` 。

仅支持 Android 3.0 和以上

如果 minSdkVersion 为 `11` 或更高，自定义主题必须继承 Theme.Holo 主题（或者其子主题）。例如：

```
<resources>
    <!-- 为程序或者活动应用的主题样式 -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.Holo">
        <item name="android:windowActionBarOverlay">true</item>
    </style>
</resources>
```

支持 Android 2.1 和更高

如果为了兼容运行在 Android 3.0 以下版本的设备而使用了 Support 库，自定义主题必须继承 Theme.AppCompat 主题（或者其子主题）。例如：

```
<resources>
    <!-- 为程序或者活动应用的主题样式 -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.AppCompat">
        <item name="android:windowActionBarOverlay">true</item>

        <!-- 兼容支持库 -->
        <item name="windowActionBarOverlay">true</item>
    </style>
</resources>
```

注意，该主题包含两种不同的 `windowActionBarOverlay` 式样定义：一个带 `android:` 前缀，另一个不带。带前缀的适用于包含该式样的 Android 系统版本，不带前缀的适用于通过从 Support 库中读取式样的旧版本。

指定布局的顶部边距

当 action bar 启用叠加模式时，它可能会遮挡住本应保持可见状态的布局。为了确保这些布局始终位于 action bar 下部，可以使用 actionBarSize 属性来指定顶部margin或padding的高度来到达。例如：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?android:attr/actionBarSize">
    ...
</RelativeLayout>
```

如果在 action bar 中使用 Support 库，需要移除 `android:` 前缀。例如：

```
<!-- 兼容支持库 -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?attr/actionBarSize">
    ...
</RelativeLayout>
```

在这种情况下，不带前缀的 `?attr/actionBarSize` 适用于包括Android 3.0 和更高的所有版本。

兼容不同的设备

> 编写:Lin-H - 原文:http://developer.android.com/training/basics/supporting-devices/index.html

全世界的Android设备有着各种各样的大小和尺寸。通过各种各样的设备类型，能使我们通过自己的app接触到广大的用户群体。为了能在各种Android平台上使用，我们的app需要兼容各种不同的设备类型。某些例如语言，屏幕尺寸，Android的系统版本等重要的变量因素需要重点考虑。

本课程会教我们如何使用基础的平台功能，利用替代资源和其他功能，使app仅用一个app程序包(APK)，就能向用Android兼容设备的用户提供最优的用户体验。

Lessons

- 适配不同的语言

  学习如何使用字符串替代资源实现支持多国语言。

- 适配不同的屏幕

  学习如何根据不同尺寸分辨率的屏幕来优化用户体验。

- 适配不同的系统版本

  学习如何在使用新的用户编程接口(API)时向下兼容旧版本Android。

适配不同的语言

> 编写:Lin-H - 原文:http://developer.android.com/training/basics/supporting-devices/languages.html

把UI中的字符串存储在外部文件，通过代码提取，这是一种很好的做法。Android可以通过工程中的资源目录轻松实现这一功能。

如果使用Android SDK Tools(详见创建Android项目(Creating an Android Project))来创建工程，则在工程的根目录会创建一个 `res/` 的目录，目录中包含所有资源类型的子目录。其中包含工程的默认文件比如 `res/values/strings.xml` ，用于保存字符串值。

创建区域设置目录及字符串文件

为支持多国语言，在 `res/` 中创建一个额外的 `values` 目录以连字符和ISO国家代码结尾命名，比如 `values-es/` 是为语言代码为"es"的区域设置的简单的资源文件的目录。Android会在运行时根据设备的区域设置，加载相应的资源。详见Providing Alternative Resources。

若决定支持某种语言，则需要创建资源子目录和字符串资源文件，例如:

```
MyProject/
    res/
       values/
           strings.xml
       values-es/
           strings.xml
       values-fr/
           strings.xml
```

添加不同区域语言的字符串值到相应的文件。

Android系统运行时会根据用户设备当前的区域设置，使用相应的字符串资源。

例如，下面列举了几个不同语言对应不同的字符串资源文件。

英语(默认区域语言)， `/values/strings.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">My Application</string>
    <string name="hello_world">Hello World!</string>
</resources>
```

西班牙语， `/values-es/strings.xml` ：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mi Aplicación</string>
    <string name="hello_world">Hola Mundo!</string>
</resources>
```

法语， `/values-fr/strings.xml` ：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mon Application</string>
    <string name="hello_world">Bonjour le monde !</string>
</resources>
```

> **Note**：可以在任何资源类型中使用区域修饰词(或者任何配置修饰符)，比如为bitmap提供本地化的版本，更多信息见Localization。

使用字符资源

我们可以在源代码和其他XML文件中通过 `<string>` 元素的 `name` 属性来引用自己的字符串资源。

在源代码中可以通过 `R.string.<string_name>` 语法来引用一个字符串资源，很多方法都可以通过这种方式来接受字符串。

例如:

```
// Get a string resource from your app's Resources
String hello = getResources().getString(R.string.hello_world);

// Or supply a string resource to a method that requires a string
TextView textView = new TextView(this);
textView.setText(R.string.hello_world);
```

在其他XML文件中，每当XML属性要接受一个字符串值时，你都可以通过 `@string/<string_name>` 语法来引用字符串资源。

例如:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

适配不同的屏幕

> 编写:Lin-H - 原文:http://developer.android.com/training/basics/supporting-devices/screens.html

Android用尺寸和分辨率这两种常规属性对不同的设备屏幕加以分类。我们应该想到自己的app会被安装在各种屏幕尺寸和分辨率的设备中。这样，app中就应该包含一些可选资源，针对不同的屏幕尺寸和分辨率，来优化其外观。

- 有4种普遍尺寸：小(small)，普通(normal)，大(large)，超大(xlarge)
- 4种普遍分辨率：低精度(ldpi), 中精度(mdpi), 高精度(hdpi), 超高精度(xhdpi)

声明针对不同屏幕所用的layout和bitmap，必须把这些可选资源放置在独立的目录中，这与适配不同语言时的做法类似。

同样要注意屏幕的方向(横向或纵向)也是一种需要考虑的屏幕尺寸变化，因此许多app会修改layout，来针对不同的屏幕方向优化用户体验。

创建不同的layout

为了针对不同的屏幕去优化用户体验，我们需要为每一种将要支持的屏幕尺寸创建唯一的XML文件。每一种layout需要保存在相应的资源目录中，目录以 `-<screen_size>` 为后缀命名。例如，对大尺寸屏幕(large screens)，一个唯一的layout文件应该保存在 `res/layout-large/` 中。

> **Note**:为了匹配合适的屏幕尺寸Android会自动地测量我们的layout文件。所以不需要因不同的屏幕尺寸去担心UI元素的大小，而应该专注于layout结构对用户体验的影响。(比如关键视图相对于同级视图的尺寸或位置)

例如，这个工程包含一个默认layout和一个适配大屏幕的layout：

```
MyProject/
    res/
        layout/
            main.xml
        layout-large/
            main.xml
```

layout文件的名字必须完全一样，为了对相应的屏幕尺寸提供最优的UI，文件的内容不同。

如平常一样在app中简单引用：

```
@Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);
}
```

系统会根据app所运行的设备屏幕尺寸，在与之对应的layout目录中加载layout。更多关于Android如何选择恰当资源的信息，详见Providing Resources。

另一个例子，这一个工程中有为适配横向屏幕的layout:

```
MyProject/
    res/
        layout/
            main.xml
        layout-land/
            main.xml
```

默认的， `layout/main.xml` 文件用作竖屏的layout。

如果想给横屏提供一个特殊的layout，也适配于大屏幕，那么则需要使用 `large` 和 `land` 修饰符。

```
 MyProject/
    res/
        layout/              # default (portrait)
            main.xml
        layout-land/         # landscape
            main.xml
        layout-large/        # large (portrait)
            main.xml
        layout-large-land/   # large landscape
            main.xml
```

> **Note**:Android 3.2及以上版本支持定义屏幕尺寸的高级方法，它允许我们根据屏幕最小长度和宽度，为各种屏幕尺寸指定与密度无关的layout资源。这节课程不会涉及这一新技术，更多信息详见Designing for Multiple Screens。

创建不同的bitmap

我们应该为4种普遍分辨率:低，中，高，超高精度，都提供相适配的bitmap资源。这能使我们的app在所有屏幕分辨率中都能有良好的画质和效果。

要生成这些图像，应该从原始的矢量图像资源着手，然后根据下列尺寸比例，生成各种密度下的图像。

- xhdpi: 2.0
- hdpi: 1.5
- mdpi: 1.0 (基准)
- ldpi: 0.75

这意味着，如果针对xhdpi的设备生成了一张200x200的图像，那么应该为hdpi生成150x150,为mdpi生成100x100, 和为ldpi生成75x75的图片资源。

然后，将这些文件放入相应的drawable资源目录中:

```
MyProject/
    res/
        drawable-xhdpi/
            awesomeimage.png
        drawable-hdpi/
            awesomeimage.png
        drawable-mdpi/
            awesomeimage.png
        drawable-ldpi/
            awesomeimage.png
```

任何时候，当引用 `@drawable/awesomeimage` 时系统会根据屏幕的分辨率选择恰当的bitmap。

> **Note**:低密度(ldpi)资源是非必要的，当提供了hdpi的图像，系统会把hdpi的图像按比例缩小一半，去适配ldpi的屏幕。

更多关于为app创建图标assets的信息和指导，详见Iconography design。

适配不同的系统版本

> 编写:Lin-H - 原文:http://developer.android.com/training/basics/supporting-devices/platforms.html

新的Android版本会为我们的app提供更棒的APIs，但我们的app仍应支持旧版本的Android，直到更多的设备升级到新版本为止。这节课程将展示如何在利用新的APIs的同时仍支持旧版本Android。

Platform Versions的控制面板会定时更新，通过统计访问Google Play Store的设备数量，来显示运行每个版本的安卓设备的分布。一般情况下，在更新app至最新Android版本时，最好先保证新版的app可以支持90%的设备使用。

> **Tip**:为了能在几个Android版本中都能提供最好的特性和功能，应该在我们的app中使用Android Support Library，它能使我们的app能在旧平台上使用最近的几个平台的APIs。

指定最小和目标API级别

AndroidManifest.xml文件中描述了我们的app的细节及app支持哪些Android版本。具体来说，`<uses-sdk>` 元素中的 `minSdkVersion` 和 `targetSdkVersion` 属性，标明在设计和测试app时，最低兼容API的级别和最高适用的API级别(这个最高的级别是需要通过我们的测试的)。例如：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ... >
    <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15" />
    ...
</manifest>
```

随着新版本Android的发布，一些风格和行为可能会改变，为了能使app能利用这些变化，而且能适配不同风格的用户的设备，我们应该将 `targetSdkVersion` 的值尽量的设置与最新可用的Android版本匹配。

运行时检查系统版本

Android在Build常量类中提供了对每一个版本的唯一代号，在我们的app中使用这些代号可以建立条件，保证依赖于高级别的API的代码，只会在这些API在当前系统中可用时，才会执行。

```
private void setUpActionBar() {
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        ActionBar actionBar = getActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    }
}
```

> **Note**:当解析XML资源时，Android会忽略当前设备不支持的XML属性。所以我们可以安全地使用较新版本的XML属性，而不需要担心旧版本Android遇到这些代码时会崩溃。例如如果我们设置 `targetSdkVersion="11"` ，app会在Android 3.0或更高时默认包含ActionBar。然后添加menu items到action bar时，我们需要在自己的menu XML资源中设置 `android:showAsAction="ifRoom"` 。在跨版本的XML文件中这么做是安全的，因为旧版本的Android会简单地忽略 `showAsAction` 属性(就是这样，你并不需要用到 `res/menu-v11/` 中单独版本的文件)。

使用平台风格和主题

Android提供了用户体验主题，为app提供基础操作系统的外观和体验。这些主题可以在manifest文件中被应用于app中。通过使用内置的风格和主题，我们的app自然地随着Android新版本的发布，自动适配最新的外观和体验.

使activity看起来像对话框:

```
<activity android:theme="@android:style/Theme.Dialog">
```

使activity有一个透明背景:

```
<activity android:theme="@android:style/Theme.Translucent">
```

应用在 `/res/values/styles.xml` 中定义的自定义主题:

```
<activity android:theme="@style/CustomTheme">
```

使整个app应用一个主题(全部activities)在元素中添加 `android:theme` 属性:

```
<application android:theme="@style/CustomTheme">
```

更多关于创建和使用主题，详见Styles and Themes。

管理Activity的生命周期

> 原文:http://developer.android.com/training/basics/activity-lifecycle/index.html

当用户导航、退出和返回您的应用时，应用中的 Activity 实例将在其生命周期中转换不同状态。 例如，当您的Activity初次开始时，它将出现在系统前台并接收用户焦点。 在这个过程中，Android 系统会对Activity调用一系列生命周期方法，通过这些方法，您可以设置用户界面和其他组件。 如果用户执行开始另一Activity或切换至另一应用的操作，当其进入后台（在其中Activity不再可见，但实例及其状态完整保留），系统会对您的Activity调用另外一系列生命周期方法。

在生命周期回调方法内，您可以声明用户离开和再次进入Activity时的Activity行为。比如，如果您正构建流视频播放器，当用户切换至另一应用时，您可能要暂停视频或终止网络连接。当用户返回时，您可以重新连接网络并允许用户从同一位置继续播放视频。

本课讲述每个 Activity 实例接收的重要生命周期回调方法以及您如何使用这些方法以使您的Activity按照用户预期进行并且当您的Activity不需要它们时不会消耗系统资源。

完整的**Demo**示例：ActivityLifecycle.zip

Lessons

- **启动与销毁Activity**

  学习有关Activity生命周期、用户如何启动您的应用以及如何执行基本Activity创建操作的基础知识。

- **暂停与恢复Activity**

  学习Activity暂停时（部分隐藏）和继续时的情况以及您应在这些状态变化期间执行的操作。

- **停止与重启Activity**

  学习用户完全离开您的Activity并返回到该Activity时发生的情况。

- **重新创建Activity**

  学习您的Activity被销毁时的情况以及您如何能够根据需要重新构建Activity。

启动与销毁Activity

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/activity-lifecycle/starting.html

不同于使用 `main()` 方法启动应用的其他编程范例，Android 系统会通过调用对应于其生命周期中特定阶段的特定回调方法在 Activity 实例中启动代码。 有一系列可启动Activity的回调方法，以及一系列可分解Activity的回调方法。

本课程概述了最重要的生命周期方法，并向您展示如何处理创建Activity新实例的第一个生命周期回调。

了解生命周期回调

在Activity的生命周期中，系统会按类似于阶梯金字塔的顺序调用一组核心的生命周期方法。也就是说，Activity生命周期的每个阶段就是金字塔上的一阶。 当系统创建新Activity实例时，每个回调方法会将Activity状态向顶端移动一阶。金字塔的顶端是Activity在前台运行并且用户可以与其交互的时间点。

当用户开始离开Activity时，系统会调用其他方法在金字塔中将Activity状态下移，从而销毁Activity。在有些情况下，Activity将只在金字塔中部分下移并等待（比如，当用户切换到其他应用时），Activity可从该点开始移回顶端（如果用户返回到该Activity），并在用户停止的位置继续。

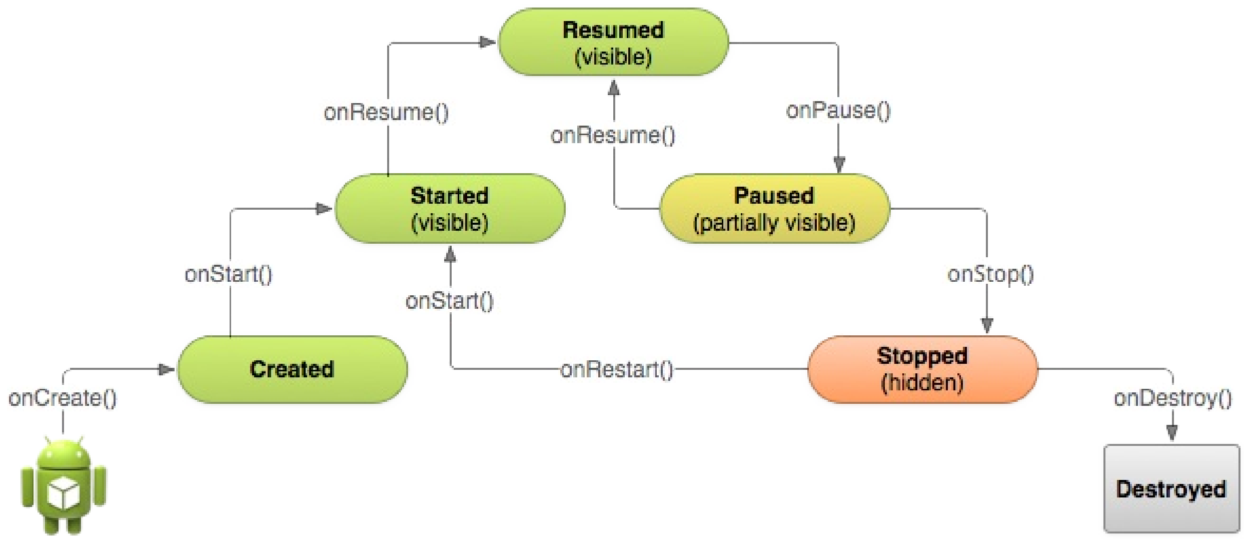


图 1.简化的Activity生命周期图示，以阶梯金字塔表示。此图示显示，对于用于将Activity朝顶端的“继续”状态移动一阶的每个回调，有一种将Activity下移一阶的回调方法。Activity还可以从“暂停”和“停止”状态回到继续状态。*

根据Activity的复杂程度，您可能不需要实现所有生命周期方法。但是，了解每个方法并实现确保您的应用按照用户期望的方式运行的方法非常重要。正确实现您的Activity生命周期方法可确保您的应用按照以下几种方式良好运行，包括：

- 如果用户在使用您的应用时接听来电或切换到另一个应用，它不会崩溃。
- 在用户未主动使用它时不会消耗宝贵的系统资源。
- 如果用户离开您的应用并稍后返回，不会丢失用户的进度。
- 当屏幕在横向和纵向之间旋转时，不会崩溃或丢失用户的进度。

正如您将要在以下课程中要学习的，有Activity会在图 1 所示不同状态之间过渡的几种情况。但是，这些状态中只有三种可以是静态。 也就是说，Activity只能在三种状态之一下存在很长时间。

- **Resumed**：在这种状态下，Activity处于前台，且用户可以与其交互。（有时也称为“运行”状态。）
- **Paused**：在这种状态下，Activity被在前台中处于半透明状态或者未覆盖整个屏幕的另一个Activity—部分阻挡。暂停的Activity不会接收用户输入并且无法执行任何代码。
- **Stopped**：在这种状态下，Activity被完全隐藏并且对用户不可见；它被视为处于后台。停止时，Activity实例及其诸如成员变量等所有状态信息将保留，但它无法执行任何代码。

其他状态（“创建”和“开始”）是瞬态，

其它状态 (**Created**与**Started**)都是短暂的瞬态，系统会通过调用下一个生命周期回调方法从这些状态快速移到下一个状态。 也就是说，在系统调用 onCreate()) 之后，它会快速调用 onStart())，紧接着快速调用 onResume())。

基本生命周期部分到此为止。现在，您将开始学习特定生命周期行为的一些知识。

指定程序首次启动的Activity

当用户从主界面点击程序图标时，系统会调用app中被声明为"launcher" (or "main") activity中的onCreate()方法。这个Activity被用来当作程序的主要进入点。

我们可以在AndroidManifest.xml中定义作为主activity的activity。

这个main activity必须在manifest使用包括 `MAIN` action 与 `LAUNCHER` category 的 `<intent-filter>` 标签来声明。例如：

```
<activity android:name=".MainActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

> **Note**:当你使用Android SDK工具来创建Android工程时，工程中就包含了一个默认的声明有这个filter的activity类。

如果程序中没有声明了MAIN action 或者LAUNCHER category的activity，那么在设备的主界面列表里面不会呈现app图标。

创建一个新的实例

大多数app包括多个activity，使用户可以执行不同的动作。不论这个activity是当用户点击应用图标创建的main activtiy还是为了响应用户行为而创建的其他activity，系统都会调用新activity实例中的onCreate()方法。

我们必须实现onCreate()方法来执行程序启动所需要的基本逻辑。例如可以在onCreate()方法中定义UI以及实例化类成员变量。

例如：下面的onCreate()方法演示了为了建立一个activity所需要的一些基础操作。如声明UI元素，定义成员变量，配置UI等。*(onCreate里面尽量少做事情，避免程序启动太久都看不到界面)*

```
TextView mTextView; // Member variable for text view in the layout

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);

    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

> **Caution**：用SDK_INT来避免旧的系统调用了只在Android 2.0（API level 5）或者更新的系统可用的方法（上述if条件中的代码）。旧的系统调用了这些方法会抛出一个运行时异常。

一旦onCreate 操作完成，系统会迅速调用onStart() 与onResume()方法。我们的activity不会在Created或者Started状态停留。技术上来说, activity在onStart()被调用后开始被用户可见，但是 onResume()会迅速被执行使得activity停留在Resumed状态，直到一些因素发生变化才会改变这个状态。例如接收到一个来电，用户切换到另外一个activity，或者是设备屏幕关闭。

在后面的课程中，我们将看到其他方法是如何使用的，onStart() 与 onResume()在用户从Paused或Stopped状态中恢复的时候非常有用。

> **Note:** onCreate() 方法包含了一个参数叫做savedInstanceState，这将会在后面的课程 - 重新创建activity涉及到。

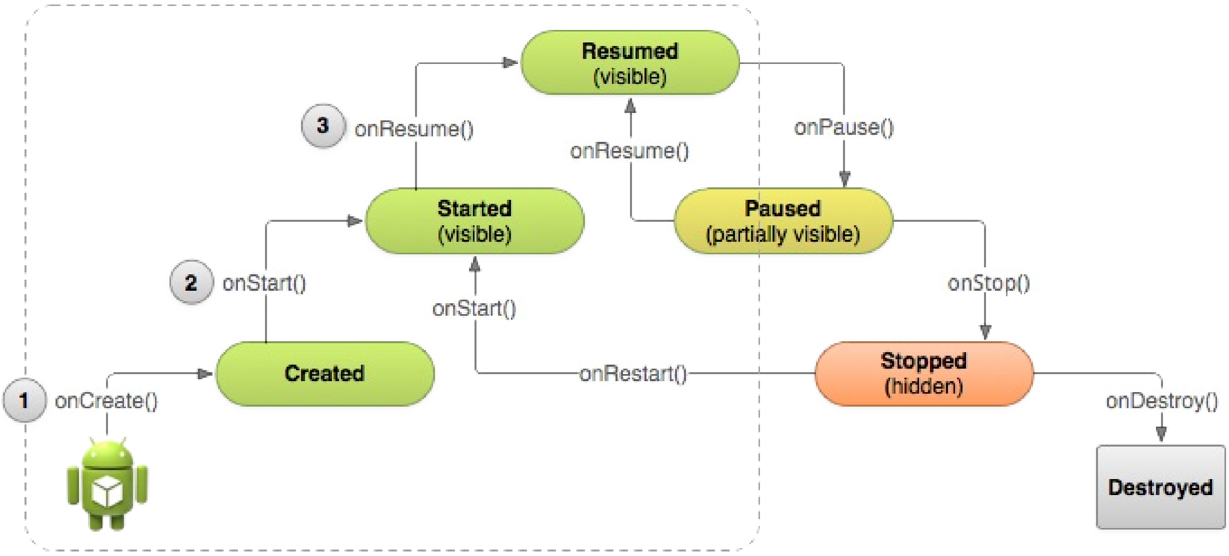


Figure 2. 上图显示了onCreate(), onStart() 和 onResume()是如何执行的。当这三个顺序执行的回调函数完成后，activity会到达Resumed状态。

销毁Activity

activity的第一个生命周期回调函数是 onCreate(),它最后一个回调是onDestroy().当收到需要将该activity彻底移除的信号时，系统会调用这个方法。

大多数 app并不需要实现这个方法，因为局部类的references会随着activity的销毁而销毁，并且我们的activity应该在onPause()与onStop()中执行清除activity资源的操作。然而，如果activity含有在onCreate调用时创建的后台线程，或者是其他有可能导致内存泄漏的资源，则应该在OnDestroy()时进行资源清理，杀死后台线程。

```
@Override
public void onDestroy() {
    super.onDestroy();  // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

> **Note:** 除非程序在onCreate()方法里面就调用了finish()方法，系统通常是在执行了onPause()与onStop() 之后再调用onDestroy() 。在某些情况下，例如我们的activity只是做了一个临时的逻辑跳转的功能，它只是用来决定跳转到哪一个activity，这样的话，需要在onCreate里面调用finish方法，这样系统会直接调用onDestory，跳过生命周期中的其他方法。

暂停与恢复Activity

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/activity-lifecycle/pausing.html

在正常使用app时，前端的activity有时会被其他可见的组件阻塞(obstructed)，从而导致当前的activity进入Pause状态。例如，当打开一个半透明的activity时(例如以对话框的形式)，之前的activity会被暂停。 只要之前的activity仍然被部分可见，这个activity就会一直处于Paused状态。

然而，一旦之前的activity被完全阻塞并不可见时，则其会进入Stop状态(将在下一小节讨论)。

activity一旦进入paused状态，系统就会调用activity中的onPause()方法, 该方法中可以停止不应该在暂停过程中执行的操作，如暂停视频播放；或者保存那些有可能需要长期保存的信息。如果用户从暂停状态回到当前activity，系统应该恢复那些数据并执行onResume()方法。

> **Note:** 当我们的activity收到调用onPause()的信号时，那可能意味者activity将被暂停一段时间，并且用户很可能回到我们的activity。然而，那也是用户要离开我们的activtiy的第一个信号。

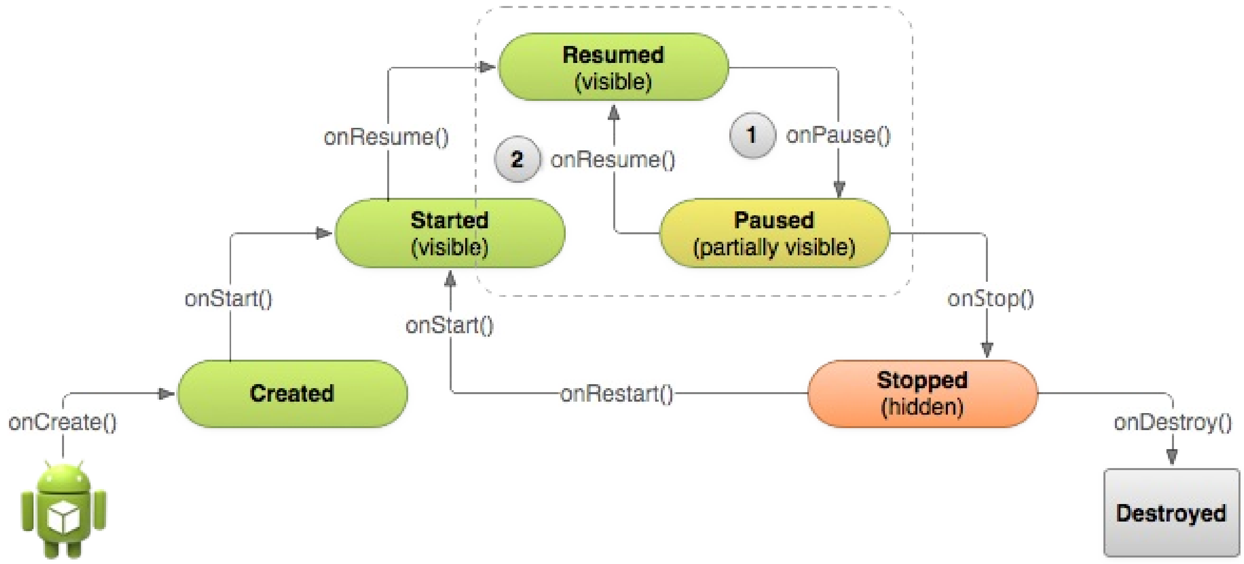


Figure 1. 当一个半透明的activity阻塞activity时，系统会调用onPause()方法并且这个activity会停留在Paused 状态(1). 如果用户在这个activity还是在Paused 状态时回到这个activity，系统则会调用它的onResume() (2).

暂停Activity

当系统调用activity中的onPause()，从技术上讲，意味着activity仍然处于部分可见的状态.但更多时候意味着用户正在离开这个activity，并马上会进入Stopped state. 通常应该在onPause()回调方法里面做以下事情:

- 停止动画或者是其他正在运行的操作，那些都会导致CPU的浪费.
- 提交在用户离开时期待保存的内容(例如邮件草稿).
- 释放系统资源，例如broadcast receivers, sensors (比如GPS), 或者是其他任何会影响到电量的资源。

例如, 如果程序使用Camera,onPause()会是一个比较好的地方去做那些释放资源的操作。

```
@Override
public void onPause() {
    super.onPause();  // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release()
        mCamera = null;
    }
}
```

通常，不应该使用onPause()来保存用户改变的数据 (例如填入表格中的个人信息) 到永久存储(File或者DB)上。仅仅当确认用户期待那些改变能够被自动保存的时候(例如正在撰写邮件草稿)，才把那些数据存到永久存储 。但是，我们应该避免在onPause()时执行CPU-intensive 的工作，例如写数据到DB，因为它会导致切换到下一个activity变得缓慢(应该把那些heavy-load的工作放到onStop()去做)。

如果activity实际上是要被Stop，那么我们应该为了切换的顺畅而减少在OnPause()方法里面的工作量。

> **Note:** 当activity处于暂停状态，Activity实例是驻留在内存中的，并且在activity 恢复的时候重新调用。我们不需要在恢复到Resumed状态的一系列回调方法中重新初始化组件。

恢复activity

当用户从Paused状态恢复activity时，系统会调用onResume()方法。

请注意，系统每次调用这个方法时，activity都处于前台，包括第一次创建的时候。所以，应该实现onResume()来初始化那些在onPause方法里面释放掉的组件，并执行那些activity每次进入Resumed state都需要的初始化动作 (例如开始动画与初始化那些只有在获取用户焦点时才需要的组件)

下面的onResume()的例子是与上面的onPause()例子相对应的。

```java
@Override
public void onResume() {
    super.onResume();  // Always call the superclass method first

    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

停止与重启Activity

> 编写:kesenhoo - 原文: http://developer.android.com/training/basics/activity-lifecycle/stopping.html

恰当的停止与重启我们的activity是很重要的，在activity生命周期中，他们能确保用户感知到程序的存在并不会丢失他们的进度。在下面一些关键的场景中会涉及到停止与重启：

- 用户打开最近使用app的菜单并从我们的app切换到另外一个app，这个时候我们的app是被停止的。如果用户通过手机主界面的启动程序图标或者最近使用程序的窗口回到我们的app，那么我们的activity会重启。
- 用户在我们的app里面执行启动一个新activity的操作，当前activity会在第二个activity被创建后stop。如果用户点击back按钮，第一个activtiy会被重启。
- 用户在使用我们的app时接收到一个来电通话.

Activity类提供了onStop()与onRestart()方法来允许在activity停止与重启时进行调用。不同于暂停状态的部分阻塞UI，停止状态是UI不再可见并且用户的焦点转移到另一个activity中.

> **Note:** 因为系统在activity停止时会在内存中保存Activity的实例，所以有时不需要实现onStop(),onRestart()甚至是onStart()方法. 因为大多数的activity相对比较简单，activity会自己停止与重启，我们只需要使用onPause()来停止正在运行的动作并断开系统资源链接。

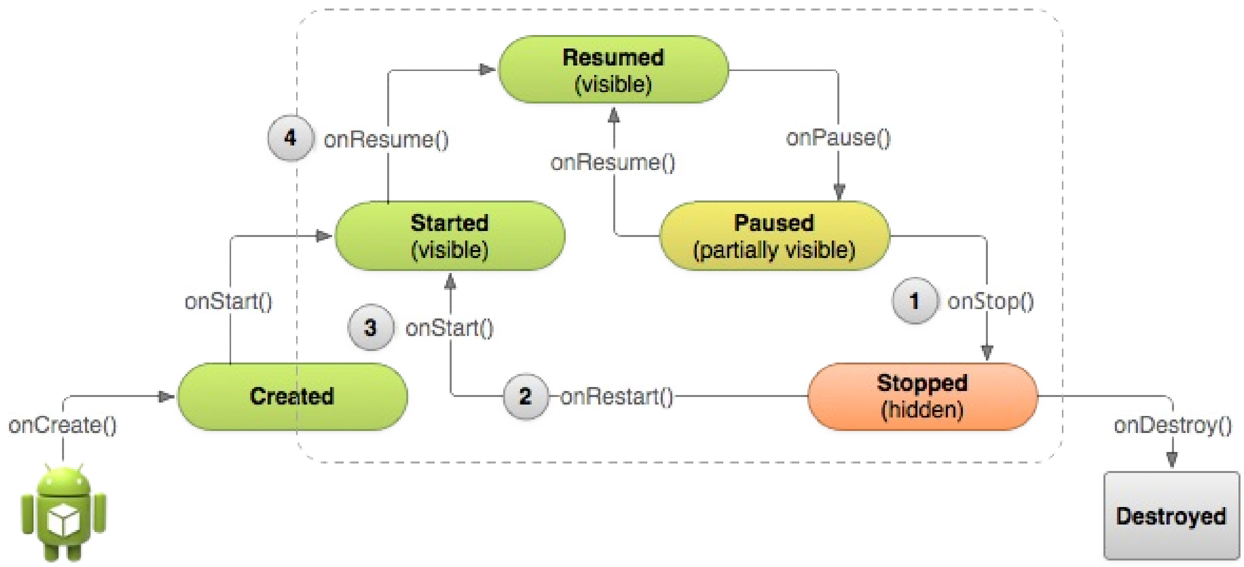


Figure 1. 上图显示：当用户离开我们的activity时，系统会调用onStop()来停止activity (1). 这个时候如果用户返回，系统会调用onRestart()(2), 之后会迅速调用onStart()(3)与onResume()(4). 请注意：无论什么原因导致activity停止，系统总是会在onStop()之前调用onPause()方法。

停止activity

当activity调用onStop()方法, activity不再可见，并且应该释放那些不再需要的所有资源。一旦activity停止了，系统会在需要内存空间时摧毁它的实例(和栈结构有关，通常*back*操作会导致前一个*activity*被销毁)。极端情况下，系统会直接杀死我们的app进程，并不执行activity的onDestroy()回调方法, 因此我们需要使用onStop()来释放资源，从而避免内存泄漏。*(这点需要注意)*

尽管onPause()方法是在onStop()之前调用，我们应该使用onStop()来执行那些CPU intensive的shut-down操作，例如往数据库写信息。

例如，下面是一个在onStop()的方法里面保存笔记草稿到persistent storage的示例:

```
@Override
protected void onStop() {
    super.onStop();  // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
            mUri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
            );
}
```

activity已经停止后，Activity对象会保存在内存中，并在activity resume时被重新调用。我们不需要在恢复到Resumed state状态前重新初始化那些被保存在内存中的组件。系统同样保存了每一个在布局中的视图的当前状态，如果用户在EditText组件中输入了text，它会被保存，因此不需要保存与恢复它。

> **Note:** 即使系统会在activity stop时停止这个activity，它仍然会保存View对象的状态(比如EditText中的文字) 到一个Bundle中，并且在用户返回这个activity时恢复它们(下一小节会介绍在activity销毁与重新建立时如何使用Bundle来保存其他数据的状态).

启动与重启activity

当activity从Stopped状态回到前台时，它会调用onRestart().系统再调用onStart()方法，onStart()方法会在每次activity可见时都会被调用。onRestart()方法则是只在activity从stopped状态恢复时才会被调用，因此我们可以使用它来执行一些特殊的恢复(restoration)工作，请注意之前是被stopped而不是destrory。

使用onRestart()来恢复activity状态是不太常见的，因此对于这个方法如何使用没有任何的guidelines。然而，因为onStop()方法应该做清除所有activity资源的操作，我们需要在重启activtiy时重新实例化那些被清除的资源，同样, 我们也需要在activity第一次创建时实例化那些资源。介于上面的原因，应该使用onStart()作为onStop()所对应方法。因为系统会在创建activity与从停止状态重启activity时都会调用onStart()。也就是说，我们在onStop里面做了哪些清除的操作，就该在onStart里面重新把那些清除掉的资源重新创建出来。

例如：因为用户很可能在回到这个activity之前已经过了很长一段时间，所以onStart()方法是一个比较好的地方来验证某些必须的系统特性是否可用。

```
@Override
protected void onStart() {
    super.onStart();  // Always call the superclass method first

    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
            (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDE
R);

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent

        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
}

@Override
protected void onRestart() {
    super.onRestart();  // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

当系统Destory我们的activity，它会为activity调用onDestroy()方法。因为我们会在onStop方法里面做释放资源的操作，那么onDestory方法则是我们最后去清除那些可能导致内存泄漏的地方。因此需要确保那些线程都被destroyed并且所有的操作都被停止。

重新创建Activity

> 编写:kesenhoo - 原文: http://developer.android.com/training/basics/activity-lifecycle/recreating.html

有几个场景中，Activity是由于正常的程序行为而被Destory的。例如当用户点击返回按钮或者是Activity通过调用finish()来发出停止信号。系统也有可能会在Activity处于stop状态且长时间不被使用，或者是在前台activity需要更多系统资源的时关闭后台进程，以图获取更多的内存。

当Activity是因为用户点击Back按钮或者是activity通过调用finish()结束自己时，系统就丢失了对Activity实例的引用，因为这一行为意味着不再需要这个activity了。然而，如果因为系统资源紧张而导致Activity的Destory， 系统会在用户回到这个Activity时有这个Activity存在过的记录，系统会使用那些保存的记录数据（描述了当Activity被Destory时的状态）来重新创建一个新的Activity实例。那些被系统用来恢复之前状态而保存的数据被叫做 "instance state" ，它是一些存放在Bundle对象中的key-value pairs。(请注意这里的描述，这对理解*onSaveInstanceState*执行的时刻很重要)

> **Caution:** 你的Activity会在每次旋转屏幕时被destroyed与recreated。当屏幕改变方向时，系统会Destory与Recreate前台的activity，因为屏幕配置被改变，你的Activity可能需要加载另一些替代的资源(例如layout).

默认情况下, 系统使用 Bundle 实例来保存每一个View(视图)对象中的信息(例如输入EditText中的文本内容)。因此，如果Activity被destroyed与recreated, 则layout的状态信息会自动恢复到之前的状态。然而，activity也许存在更多你想要恢复的状态信息，例如记录用户Progress的成员变量(member variables)。

> **Note:** 为了使Android系统能够恢复Activity中的View的状态，每个**View**都必须有一个唯一**ID**，由android:id定义。

为了可以保存额外更多的数据到saved instance state。在Activity的生命周期里面存在一个额外的回调函数，你必须重写这个函数。该回调函数并没有在前面课程的图片示例中显示。这个方法是onSaveInstanceState() ，当用户离开Activity时，系统会调用它。当系统调用这个函数时，系统会在Activity被异常Destory时传递 Bundle 对象，这样我们就可以增加额外的信息到Bundle中并保存到系统中。若系统在Activity被Destory之后想重新创建这个Activity实例时，之前的Bundle对象会(系统)被传递到你我们activity的onRestoreInstanceState()方法与onCreate() 方法中。

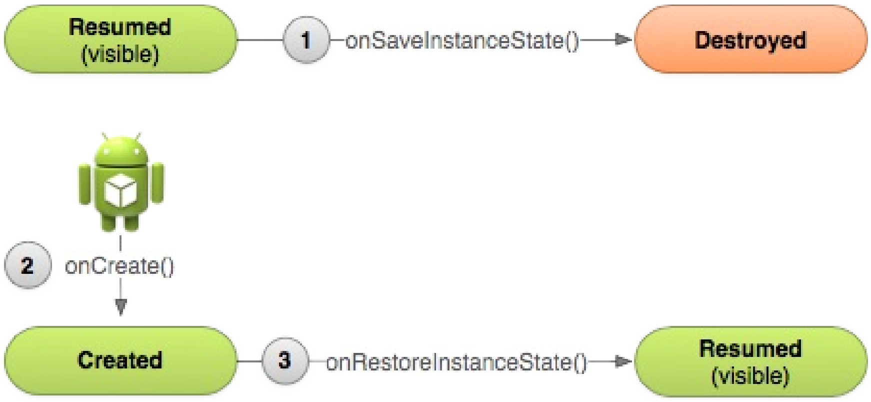


Figure 2. 当系统开始停止Activity时，只有在Activity实例会需要重新创建的情况下才会调用到onSaveInstanceState() (1) ，在这个方法里面可以指定额外的状态数据到Bunde中。如果这个Activity被destroyed然后这个实例又需要被重新创建时，系统会传递在 (1) 中的状态数据到onCreate() (2) 与 onRestoreInstanceState()(3).

(通常来说，跳转到其他的activity或者是点击Home都会导致当前的activity执行onSaveInstanceState，因为这种情况下的activity都是有可能会被destory并且是需要保存状态以便后续恢复使用的，而从跳转的activity点击back回到前一个activity，那么跳转前的activity是执行退栈的操作，所以这种情况下是不会执行onSaveInstanceState的，因为这个activity不可能存在需要重建的操作)

保存Activity状态

当我们的activity开始Stop，系统会调用 onSaveInstanceState() ，Activity可以用键值对的集合来保存状态信息。这个方法会默认保存Activity视图的状态信息，如在 EditText 组件中的文本或 ListView 的滑动位置。

为了给Activity保存额外的状态信息，你必须实现onSaveInstanceState() 并增加key-value pairs到 Bundle 对象中，例如：

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

> **Caution**: 必须要调用 onSaveInstanceState() 方法的父类实现，这样默认的父类实现才能保存视图状态的信息。

恢复Activity状态

当Activity从Destory中重建，我们可以从系统传递的Activity的Bundle中恢复保存的状态。onCreate() 与 onRestoreInstanceState() 回调方法都接收到了同样的Bundle，里面包含了同样的实例状态信息。

由于 onCreate() 方法会在第一次创建新的Activity实例与重新创建之前被Destory的实例时都被调用，我们必须在尝试读取 Bundle 对象前检测它是否为null。如果它为null，系统则是创建一个新的Activity实例，而不是恢复之前被Destory的Activity。

下面是一个示例：演示在onCreate方法里面恢复一些数据：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

我们也可以选择实现 onRestoreInstanceState() ，而不是在onCreate方法里面恢复数据。**onRestoreInstanceState()**方法会在 **onStart()** 方法之后执行. 系统仅仅会在存在需要恢复的状态信息时才会调用 **onRestoreInstanceState()** ，因此不需要检查 **Bundle** 是否为**null**。

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

> **Caution**: 与上面保存一样，总是需要调用onRestoreInstanceState()方法的父类实现，这样默认的父类实现才能保存视图状态的信息。更多关于运行时状态改变引起的recreate我们的activity。请参考Handling Runtime Changes.

使用Fragment建立动态UI

> 编写：fastcome1985 - 原文：https://developer.android.com/training/basics/fragments/index.html

为了在 Android 上为用户提供动态的、多窗口的交互体验，需要将 UI 组件和 Activity 操作封装成模块进行使用，这样我们就可以在 Activity 中对这些模块进行切入切出操作。可以用 Fragment 创建这些模块，Fragment 就像一个嵌套的 Activity，拥有自己的布局（Layout）并管理自己的生命周期。

Fragment 定义了自己的布局后，它可以在 Activity 中与其他 Fragment 生成不同的组合，从而为不同的屏幕尺寸生成不同的布局（小屏幕一次也许只能显示一个 Fragment，大屏幕则可以显示更多）。

本章将展示如何用 Fragment 创建动态界面，并在不同屏幕尺寸的设备上优化 APP 的用户体验。本章内容支持 Android 1.6 以上的设备。

（完整的 Demo 示例：FragmentBasics.zip）

Lessons

- **创建 Fragment**

  学习如何创建 Fragment，以及实现其生命周期内的基本功能。

- **构建有弹性的 UI**

  学习如何针对不同的屏幕尺寸用 Fragment 构建不同的布局。

- **与其他 Fragment 交互**

  学习如何在 Fragment 与 Activity 或多个 Fragment 间进行交互。

创建 Fragment

> 编写：fastcome1985 - 原文：https://developer.android.com/training/basics/fragments/creating.html

可以把 Fragment 想象成 Activity 的模块，它拥有自己的生命周期、接收输入事件，可以在 Acvitity 运行过程中添加或者移除（有点像"子 Activity"，可以在不同的 Activity 里重复使用）。这一课教我们将学习继承 Support Library 中的 Fragment，使 APP 在 Android 1.6 这样的低版本上仍能保持兼容。

在开始之前，必须在项目中先引用 Support Library。如果你从未使用过 Support Library，可根据文档 设置 Support Library 在项目中使用 **v4** 库。当然，也可以使用包含 APP Bar 的 **v7 appcompat** 库。该库兼容 Android 2.1 (API level 7)，同时也包含了 Fragment API。

创建 Fragment 类

首先从 Fragment 继承并创建 Fragment，然后在关键的生命周期方法中插入代码（就和在处理 Activity 时一样）。

其中一个区别是：创建 Fragment 时，必须重写 onCreateView() 回调方法来定义布局。事实上，这是唯一一个为使 Fragment 运行起来需要重写的回调方法。比如，下面是一个自定义布局的示例 Fragment：

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;

public class ArticleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // 拉伸该 Fragment 的布局
        return inflater.inflate(R.layout.article_view, container, false);
    }
}
```

和 Activity 一样，当 Fragment 从 Activity 添加或者移除、或 Activity 生命周期发生变化时，Fragment 通过生命周期回调函数管理其状态。例如，当 Activity 的 onPause() 被调用时，它内部所有 Fragment 的 onPause() 方法也会被触发。

更多关于 Fragment 的声明周期和回调方法，详见 Fragments 开发指南.

用 XML 将 Fragment 添加到 Activity

Fragments 是可重用的、模块化的 UI 组件。每个 Fragment 实例都必须与一个 FragmentActivity 关联。我们可以在 Activity 的 XML 布局文件中逐个定义 Fragment 来实现这种关联。

> 注：FragmentActivity 是 Support Library 提供的一种特殊 Activity，用于处理 API 11 版本以下的 Fragment。如果我们 APP 中的最低版本大于等于 11，则可以使用普通的 Activity。

以下是一个 XML 布局的例子：当屏幕被认为是 "large"（用目录名称中的 `large` 字符来区分）时，它在布局中增加了两个 Fragment。

res/layout-large/news_articles.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <fragment android:name="com.example.android.fragments.HeadlinesFragment"
              android:id="@+id/headlines_fragment"
              android:layout_weight="1"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

    <fragment android:name="com.example.android.fragments.ArticleFragment"
              android:id="@+id/article_fragment"
              android:layout_weight="2"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

</LinearLayout>
```

> 提示：更多关于不同屏幕尺寸创建不同布局的信息，请阅读 兼容不同屏幕尺寸。

然后将这个布局文件用到 Activity 中。

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);
    }
}
```

如果使用 v7 appcompat 库，Activity 应该改为继承自 AppCompatActivity，AppCompatActivity 是 FragmentActivity 的子类（更多关于这方面的内容，请阅读 添加 App Bar）。

> 注： 当通过 XML 布局文件的方式将 Fragment 添加进 Activity 时，Fragment 是不能被动态移除的。如果想要在用户交互的时候把 Fragment 切入与切出，必须在 Activity 启动后，再将 Fragment 添加进 Activity。这部分内容将在下节课阐述。

建立灵活动态的 UI

> 编写：fastcome1985 - 原文：https://developer.android.com/training/basics/fragments/fragment-ui.html

在设计支持各种屏幕尺寸的应用时，你可以在不同的布局配置中重复使用 Fragment，以便根据相应的屏幕空间提供更出色的用户体验。

例如，一次只显示一个 Fragment 可能就很适合手机这种单窗格界面，但在平板电脑上，你可能需要设置并列的 Fragment，因为平板电脑的屏幕尺寸较宽阔，可向用户显示更多信息。

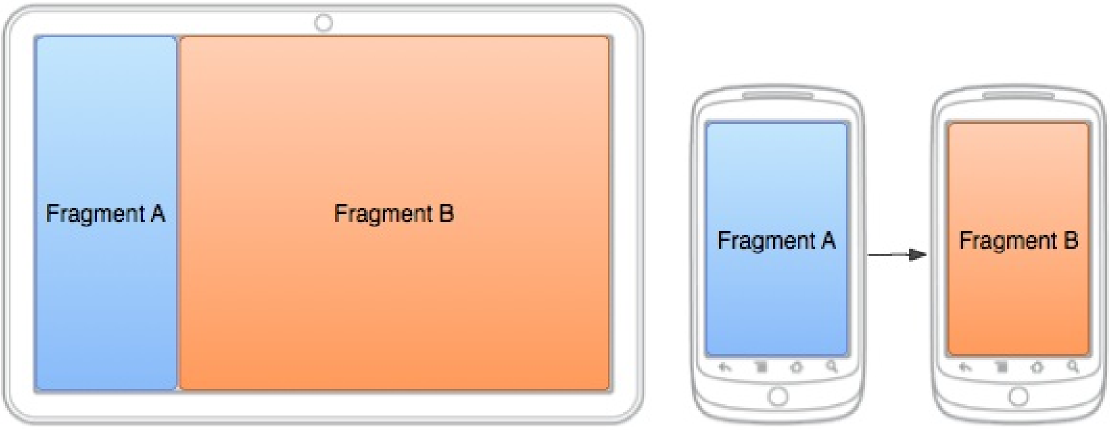


图1： 两个 Fragment，显示在不同尺寸屏幕上同一 Activity 的不同配置中。在较宽阔的屏幕上，两个 Fragment 可并列显示；在手机上，一次只能显示一个 Fragment，因此必须在用户导航时更换 Fragment。

利用 FragmentManager 类提供的方法，你可以在运行时添加、移除和替换 Activity 中的 Fragment，以便为用户提供一种动态体验。

在运行时向 Activity 添加 Fragment

你可以在 Activity 运行时向其添加 Fragment，而不用像 上一课 中介绍的那样，使用 `<fragment>` 元素在布局文件中为 Activity 定义 Fragment。如果你打算在 Activity 运行周期内更改 Fragment，就必须这样做。

要执行添加或移除 Fragment 等事务，你必须使用 FragmentManager 创建一个 FragmentTransaction，后者可提供用于执行添加、移除、替换以及其他 Fragment 事务的 API。

如果 Activity 中的 Fragment 可以移除和替换，你应在调用 Activity 的 onCreate() 方法期间为 Activity 添加初始 Fragment(s)。

在处理 Fragment（特别是在运行时添加的 Fragment）时，请谨记以下重要规则：必须在布局中为 Fragment 提供 View 容器，以便保存 Fragment 的布局。

下面是 上一课 所示布局的替代布局，这种布局一次只会显示一个 Fragment。要用一个 Fragment 替换另一个 Fragment，Activity 的布局中需要包含一个作为 Fragment 容器的空 FrameLayout。

请注意，该文件名与上一课中布局文件的名称相同，但布局目录没有 `large` 这一限定符。因此，此布局会在设备屏幕小于“large”的情况下使用，原因是尺寸较小的屏幕不适合同时显示两个 Fragment。

res/layout/news_articles.xml:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

在 Activity 中，用 Support Library API 调用 getSupportFragmentManager() 以获取 FragmentManager，然后调用 beginTransaction() 创建 FragmentTransaction，然后调用 add() 添加 Fragment。

你可以使用同一个 FragmentTransaction 对 Activity 执行多 Fragment 事务。当你准备好进行更改时，必须调用 commit()。

例如，下面介绍了如何为上述布局添加 Fragment：

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);

        // 确认 Activity 使用的布局版本包含 fragment_container FrameLayout
        if (findViewById(R.id.fragment_container) != null) {

            // 不过，如果我们要从先前的状态还原，则无需执行任何操作而应返回，否则
            // 就会得到重叠的 Fragment。
            if (savedInstanceState != null) {
                return;
            }

            // 创建一个要放入 Activity 布局中的新 Fragment
            HeadlinesFragment firstFragment = new HeadlinesFragment();

            // 如果此 Activity 是通过 Intent 发出的特殊指令来启动的，
            // 请将该 Intent 的 extras 以参数形式传递给该 Fragment
            firstFragment.setArguments(getIntent().getExtras());

            // 将该 Fragment 添加到“fragment_container” FrameLayout 中
            getSupportFragmentManager().beginTransaction()
                    .add(R.id.fragment_container, firstFragment).commit();
        }
    }
}
```

由于该 Fragment 已在运行时添加到 FrameLayout 容器中，而不是在 Activity 布局中通过 `<fragment>` 元素进行定义，因此该 Activity 可以移除和替换这个 Fragment。

用一个 Fragment 替换另一个 Fragment

替换 Fragment 的步骤与添加 Fragment 的步骤相似，但需要调用 replace() 方法，而非 add()。

请注意，当你执行替换或移除 Fragment 等 Fragment 事务时，最好能让用户向后导航和“撤消”所做更改。要通过 Fragment 事务允许用户向后导航，你必须调用 addToBackStack()，然后再执行 FragmentTransaction。

> 注： 当你移除或替换 Fragment 并向返回堆栈添加事务时，已移除的 Fragment 会停止（而不是销毁）。如果用户向后导航，还原该 Fragment，它会重新启动。如果你没有向返回堆栈添加事务，那么该 Fragment 在移除或替换时就会被销毁。

替换 Fragment 的示例：

```
// 创建 Fragment 并为其添加一个参数，用来指定应显示的文章
ArticleFragment newFragment = new ArticleFragment();
Bundle args = new Bundle();
args.putInt(ArticleFragment.ARG_POSITION, position);
newFragment.setArguments(args);

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

// 将 fragment_container View 中的内容替换为此 Fragment，
// 然后将该事务添加到返回堆栈，以便用户可以向后导航
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// 执行事务
transaction.commit();
```

addToBackStack() 方法可接受可选的字符串参数，来为事务指定独一无二的名称。除非你打算使用 FragmentManager.BackStackEntry API 执行高级 Fragment 操作，否则无需使用此名称。

与其他 Fragment 交互

> 编写：fastcome1985 - 原文：https://developer.android.com/training/basics/fragments/communicating.html

为了重用 Fragment UI 组件，你应该把每个 Fragment 都构建成完全自包含的、模块化的组件，即，定义它们自己的布局与行为。一旦你定义了这些可重用的 Fragment，你就可以通过应用程序逻辑让它们关联到 Activity，以实现整体的复合 UI。

通常 Fragment 之间可能会需要交互，比如基于用户事件的内容变更。所有 Fragment 之间的交互应通过与之关联的 Activity 来完成。两个 Fragment 之间不应直接交互。

定义接口

为了让 Fragment 与包含它的 Activity 进行交互，可以在 Fragment 类中定义一个接口，并在 Activity 中实现。该 Fragment 在它的 onAttach() 方法生命周期中获取该接口的实现，然后调用接口的方法，以便与 Activity 进行交互。（译注：意即，若该 Fragment 中实现了 onAttach() 方法，则会被自动调用。）

以下是 Fragment 与 Activity 交互的例子：

```
public class HeadlinesFragment extends ListFragment {
    OnHeadlineSelectedListener mCallback;

    // 容器 Activity 必须实现该接口
    // （译注："容器 Activity"意即"包含该 Fragment 的 Activity"）
    public interface OnHeadlineSelectedListener {
        public void onArticleSelected(int position);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);

        // 确认容器 Activity 已实现该回调接口。否则，抛出异常
        try {
            mCallback = (OnHeadlineSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString()
                    + " must implement OnHeadlineSelectedListener");
        }
    }

    ...
}
```

现在 Fragment 可以通过调用 `mCallback` （ `OnHeadlineSelectedListener` 接口的实例）的 `onArticleSelected()` 方法（也可以是其它方法）与 Activity 进行消息传递。

例如，当用户点击列表条目时，Fragment 中的下面的方法将被调用。Fragment 用回调接口将事件传递给父 Activity。

```
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // 向宿主 Activity 传送事件
        mCallback.onArticleSelected(position);
    }
```

实现接口

为了接收回调事件，宿主 Activity 必须实现在 Fragment 中定义的接口。

例如，下面的 Activity 实现了上面例子中的接口。

```
public static class MainActivity extends Activity
        implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // 用户从 HeadlinesFragment 选择了一篇文章的标题
        // 在这里做点什么，以显示该文章
    }
}
```

向 Fragment 传递消息

宿主 Activity 通过 findFragmentById() 获取 Fragment 的实例，然后直接调用 Fragment 的 public 方法向 Fragment 传递消息。

例如，假设上面所示的 Activity 可能包含另一个 Fragment，该 Fragment 用于展示从上面的回调方法中返回的指定的数据。在这种情况下，Activity 可以把从回调方法中接收到的信息传递到这个展示数据的 Fragment。

```
public static class MainActivity extends Activity
        implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // 用户从 HeadlinesFragment 选择了一篇文章的标题
        // 在这里做点什么，以显示该文章

        ArticleFragment articleFrag = (ArticleFragment)
                getSupportFragmentManager().findFragmentById(R.id.article_fragment);

        if (articleFrag != null) {
            // 若 articleFrag 有效，则表示我们正在处理两格布局（two-pane layout）……

            // 调用 ArticleFragment 的方法，以更新其内容
            articleFrag.updateArticleView(position);
        } else {
            // 否则，我们正在处理单格布局（one-pane layout）。此时需要 swap frags...

            // 创建 Fragment，向其传递包含被选文章的参数
            ArticleFragment newFragment = new ArticleFragment();
            Bundle args = new Bundle();
            args.putInt(ArticleFragment.ARG_POSITION, position);
            newFragment.setArguments(args);

            FragmentTransaction transaction = getSupportFragmentManager().beginTransac
tion();

            // 无论 fragment_container 视图里是什么，用该 Fragment 替换它。并将
            // 该事务添加至回栈，以便用户可以往回导航（译注：回栈，即 Back Stack。
            // 在有多个 Activity 的 APP 中，将这些 Activity 按创建次序组织起来的
            // 栈，称为回栈）
            transaction.replace(R.id.fragment_container, newFragment);
            transaction.addToBackStack(null);

            // 执行事务
            transaction.commit();
        }
    }
}
```

数据保存

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/data-storage/index.html

虽然可以在onPause()时保存一些信息以免用户的使用进度被丢失，但大多数Android app仍然是需执行保存数据的动作。大多数较好的apps都需要保存用户的设置信息，而且有一些apps必须维护大量的文件信息与DB信息。本章节将介绍Android中主要的数据存储方法，包括：

- 保存到**Preferences**

  学习使用Shared Preferences文件以Key-Value的方式保存简要的信息。

- 保存到文件

  学习保存基本的文件。

- 保存到数据库

  学习使用SQLite数据库读写数据。

保存到Preference

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/data-storage/shared-preferences.html

当有一个相对较小的key-value集合需要保存时，可以使用SharedPreferences APIs。SharedPreferences 对象指向一个保存key-value pairs的文件，并为读写他们提供了简单的方法。每个 SharedPreferences 文件均由framework管理，其既可以是私有的，也可以是共享的。 这节课会演示如何使用 SharedPreferences APIs 来存储与检索简单的数据。

> **Note**： SharedPreferences APIs 仅仅提供了读写key-value对的功能，请不要与Preference APIs相混淆。后者可以帮助我们建立一个设置用户配置的页面（尽管它实际上是使用SharedPreferences 来实现保存用户配置的)。更多关于Preference APIs的信息，请参考Settings 指南。

获取SharedPreference

我们可以通过以下两种方法之一创建或者访问shared preference 文件:

- getSharedPreferences() — 如果需要多个通过名称参数来区分的shared preference文件, 名称可以通过第一个参数来指定。可在app中通过任何一个Context 执行该方法。
- getPreferences() — 当activity仅需要一个shared preference文件时。因为该方法会检索activity下默认的shared preference文件，并不需要提供文件名称。

例：下面的示例在一个 Fragment 中被执行，它以private模式访问名为 `R.string.preference_file_key` 的shared preference文件。这种情况下，该文件仅能被我们的app访问。

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

应以与app相关的方式为shared preference文件命名，该名称应唯一。如本例中可将其命名为 `"com.example.myapp.PREFERENCE_FILE_KEY"` 。

当然，当activity仅需要一个shared preference文件时，我们可以使用getPreferences()方法：

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

> **Caution:** 如果创建了一个MODE_WORLD_READABLE或者MODE_WORLD_WRITEABLE 模式的shared preference文件，则其他任何app均可通过文件名访问该文件。

写Shared Preference

为了写 `shared preferences` 文件，需要通过执行edit()创建一个 SharedPreferences.Editor。

通过类似putInt()与putString()等方法传递keys与values，接着通过commit() 提交改变.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

读Shared Preference

为了从shared preference中读取数据，可以通过类似于 getInt() 及 getString()等方法来读取。在那些方法里面传递我们想要获取的value对应的key，并提供一个默认的value作为查找的key不存在时函数的返回值。如下：

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), default);
```

保存到文件

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/data-storage/files.html

Android使用与其他平台类似的基于磁盘的文件系统(disk-based file systems)。本课程将描述如何在Android文件系统上使用 File 的读写APIs对Andorid的file system进行读写。

File 对象非常适合于流式顺序数据的读写。如图片文件或是网络中交换的数据等。

本课程将会演示如何在app中执行基本的文件相关操作。假定读者已对linux的文件系统及java.io中标准的I/O APIs有一定认识。

存储在内部还是外部

所有的Android设备均有两个文件存储区域："internal" 与 "external" 。 这两个名称来自于早先的Android系统，当时大多设备都内置了不可变的内存（internal storage)及一个类似于SD card（external storage）这样的可卸载的存储部件。之后有一些设备将"internal" 与 "external" 都做成了不可卸载的内置存储，虽然如此，但是这一整块还是从逻辑上有被划分为"internal"与"external"的。只是现在不再以是否可卸载进行区分了。 下面列出了两者的区别：

- **Internal storage:**
  - 总是可用的
  - 这里的文件默认只能被我们的app所访问。
  - 当用户卸载app的时候，系统会把internal内该app相关的文件都清除干净。
  - Internal是我们在想确保不被用户与其他app所访问的最佳存储区域。
- **External storage:**
  - 并不总是可用的，因为用户有时会通过USB存储模式挂载外部存储器，当取下挂载的这部分后，就无法对其进行访问了。
  - 是大家都可以访问的，因此保存在这里的文件可能被其他程序访问。
  - 当用户卸载我们的app时，系统仅仅会删除external根目录（getExternalFilesDir()）下的相关文件。
  - External是在不需要严格的访问权限并且希望这些文件能够被其他app所共享或者是允许用户通过电脑访问时的最佳存储区域。

> **Tip:** 尽管app是默认被安装到internal storage的，我们还是可以通过在程序的manifest文件中声明android:installLocation 属性来指定程序安装到external storage。当某个程序的安装文件很大且用户的external storage空间大于internal storage时，用户会倾向于将该程序安装到external storage。更多安装信息见App Install Location。

获取External存储的权限

为了写数据到external storage, 必须在你manifest文件中请求WRITE_EXTERNAL_STORAGE权限：

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

> **Caution:** 目前，所有的apps都可以在不指定某个专门的权限下做读external storage的动作。但这在以后的安卓版本中会有所改变。如果我们的app只需要读的权限(不是写), 那么将需要声明 READ_EXTERNAL_STORAGE 权限。为了确保app能持续地正常工作，我们现在在编写程序时就需要声明读权限。
>
> ```
> <manifest ...>
>     <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
>     ...
> </manifest>
> ```
>
> 但是，如果我们的程序有声明**WRITE_EXTERNAL_STORAGE** 权限，那么就默认有了读的权限。

对于internal storage，我们不需要声明任何权限，因为程序默认就有读写程序目录下的文件的权限。

保存到Internal Storage

当保存文件到internal storage时，可以通过执行下面两个方法之一来获取合适的目录作为 FILE 的对象：

- getFilesDir() : 返回一个File，代表了我们app的internal目录。
- getCacheDir() : 返回一个File，代表了我们app的internal缓存目录。请确保这个目录下的文件能够在一旦不再需要的时候马上被删除，并对其大小进行合理限制，例如1MB 。系统的内部存储空间不够时，会自行选择删除缓存文件。

可以使用File() 构造器在那些目录下创建一个新的文件，如下：

```
File file = new File(context.getFilesDir(), filename);
```

同样，也可以执行openFileOutput() 获取一个 FileOutputStream 用于写文件到internal目录。如下：

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

如果需要缓存一些文件，可以使用createTempFile()。例如：下面的方法从URL中抽取了一个文件名，然后再在程序的internal缓存目录下创建了一个以这个文件名命名的文件。

```
 public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

> **Note:** 我们的app的internal storage 目录以app的包名作为标识存放在Android文件系统的特定目录下[data/data/com.example.xx]。 从技术上讲，如果文件被设置为可读的，那么其他app就可以读取该internal文件。然而，其他app需要知道包名与文件名。若没有设置为可读或者可写，其他app是没有办法读写的。因此我们只要使用了MODE_PRIVATE ，那么这些文件就不可能被其他app所访问。

保存文件到External Storage

因为external storage可能是不可用的，比如遇到SD卡被拔出等情况时。因此在访问之前应对其可用性进行检查。我们可以通过执行 getExternalStorageState()来查询external storage的状态。若返回状态为MEDIA_MOUNTED, 则可以读写。示例如下：

```
 /* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

尽管external storage对于用户与其他app是可修改的，我们可能会保存下面两种类型的文件。

- **Public files** :这些文件对与用户与其他app来说是public的，当用户卸载我们的app时，这些文件应该保留。例如，那些被我们的app拍摄的图片或者下载的文件。
- **Private files**: 这些文件完全被我们的app所私有，它们应该在app被卸载时删除。尽管由于存储在external storage，那些文件从技术上而言可以被用户与其他app所访问，但实际上那些文件对于其他app没有任何意义。因此，当用户卸载我们的app时，系统会删除其下的private目录。例如，那些被我们的app下载的缓存文件。

想要将文件以public形式保存在external storage中，请使用getExternalStoragePublicDirectory()方法来获取一个 File 对象，该对象表示存储在external storage的目录。这个方法会需要带有一个特定的参数来指定这些public的文件类型，以便于与其他public文件进行分类。参数类型包括DIRECTORY_MUSIC 或者 DIRECTORY_PICTURES. 如下:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

想要将文件以private形式保存在external storage中，可以通过执行getExternalFilesDir() 来获取相应的目录，并且传递一个指示文件类型的参数。每一个以这种方式创建的目录都会被添加到external storage封装我们app目录下的参数文件夹下（如下则是albumName）。这下面的文件会在用户卸载我们的app时被系统删除。如下示例：

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

如果刚开始的时候，没有预定义的子目录存放我们的文件，可以在 getExternalFilesDir()方法中传递 `null`. 它会返回app在external storage下的private的根目录。

请记住，getExternalFilesDir() 方法会创建的目录会在app被卸载时被系统删除。如果我们的文件想在app被删除时仍然保留，请使用getExternalStoragePublicDirectory().

无论是使用 getExternalStoragePublicDirectory() 来存储可以共享的文件，还是使用 getExternalFilesDir() 来储存那些对于我们的app来说是私有的文件，有一点很重要，那就是要使用那些类似 `DIRECTORY_PICTURES` 的API的常量。那些目录类型参数可以确保那些文件被系统正确的对待。例如，那些以 `DIRECTORY_RINGTONES` 类型保存的文件就会被系统的media scanner认为是ringtone而不是音乐。

查询剩余空间

如果事先知道想要保存的文件大小，可以通过执行getFreeSpace() or getTotalSpace() 来判断是否有足够的空间来保存文件，从而避免发生IOException。那些方法提供了当前可用的空间还有存储系统的总容量。

然而，系统并不能保证可以写入通过 `getFreeSpace()` 查询到的容量文件， 如果查询的剩余容量比我们的文件大小多几MB，或者说文件系统使用率还不足90%，这样则可以继续进行写的操作，否则最好不要写进去。

> **Note**：并没有强制要求在写文件之前去检查剩余容量。我们可以尝试先做写的动作，然后通过捕获 IOException 。这种做法仅适合于事先并不知道想要写的文件的确切大小。例如，如果在把PNG图片转换成JPEG之前，我们并不知道最终生成的图片大小是多少。

删除文件

在不需要使用某些文件的时候应删除它。删除文件最直接的方法是直接执行文件的 `delete()` 方法。

```
myFile.delete();
```

如果文件是保存在internal storage，我们可以通过 `Context` 来访问并通过执行 `deleteFile()` 进行删除

```
myContext.deleteFile(fileName);
```

> **Note:** 当用户卸载我们的app时，android系统会删除以下文件：
>
> - 所有保存到internal storage的文件。
> - 所有使用getExternalFilesDir()方式保存在external storage的文件。
>
> 然而，通常来说，我们应该手动删除所有通过 getCacheDir() 方式创建的缓存文件，以及那些不会再用到的文件。

保存到数据库

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/data-storage/databases.html

对于重复或者结构化的数据（如联系人信息）等保存到DB是个不错的主意。本课假定读者已经熟悉SQL数据库的常用操作。在Android上可能会使用到的APIs，可以从android.database.sqlite包中找到。

定义Schema与Contract

SQL中一个重要的概念是schema：一种DB结构的正式声明，用于表示database的组成结构。schema是从创建DB的SQL语句中生成的。我们会发现创建一个伴随类（companion class）是很有益的，这个类称为合约类（contract class）,它用一种系统化并且自动生成文档的方式，显示指定了schema样式。

Contract Clsss是一些常量的容器。它定义了例如URIs，表名，列名等。这个contract类允许在同一个包下与其他类使用同样的常量。 它让我们只需要在一个地方修改列名，然后这个列名就可以自动传递给整个code。

组织contract类的一个好方法是在类的根层级定义一些全局变量，然后为每一个table来创建内部类。

> **Note**：通过实现 BaseColumns 的接口，内部类可以继承到一个名为_ID的主键，这个对于Android里面的一些类似cursor adaptor类是很有必要的。这么做不是必须的，但这样能够使得我们的DB与Android的framework能够很好的相容。

例如，下面的例子定义了表名与该表的列名：

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // give it an empty constructor.
    public FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static abstract class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_ENTRY_ID = "entryid";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle";
        ...
    }
}
```

使用SQL Helper创建DB

定义好了的DB的结构之后，就应该实现那些创建与维护db和table的方法。下面是一些典型的创建与删除table的语句。

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedReaderContract.FeedEntry.TABLE_NAME + " (" +
    FeedReaderContract.FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    " )";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + TABLE_NAME_ENTRIES;
```

类似于保存文件到设备的internal storage ，Android会将db保存到程序的private的空间。我们的数据是受保护的，因为那些区域默认是私有的，不可被其他程序所访问。

在SQLiteOpenHelper类中有一些很有用的APIs。当使用这个类来做一些与db有关的操作时，系统会对那些有可能比较耗时的操作（例如创建与更新等）在真正需要的时候才去执行，而不是在app刚启动的时候就去做那些动作。我们所需要做的仅仅是执行getWritableDatabase()或者getReadableDatabase().

> **Note**：因为那些操作可能是很耗时的，请确保在background thread（AsyncTask or IntentService）里面去执行 getWritableDatabase() 或者 getReadableDatabase() 。

为了使用 SQLiteOpenHelper, 需要创建一个子类并重写onCreate(), onUpgrade()与onOpen()等callback方法。也许还需要实现onDowngrade(), 但这并不是必需的。

例如，下面是一个实现了SQLiteOpenHelper 类的例子：

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

为了访问我们的db，需要实例化 SQLiteOpenHelper的子类：

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

添加信息到DB

通过传递一个 ContentValues 对象到insert()方法：

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
         FeedReaderContract.FeedEntry.TABLE_NAME,
         FeedReaderContract.FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```

`insert()` 方法的第一个参数是table名，第二个参数会使得系统自动对那些 `ContentValues` 没有提供数据的列填充数据为 `null` ，如果第二个参数传递的是null，那么系统则不会对那些没有提供数据的列进行填充。

从DB中读取信息

为了从DB中读取数据，需要使用query()方法，传递需要查询的条件。查询后会返回一个Cursor 对象。

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedReaderContract.FeedEntry._ID,
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE,
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedReaderContract.FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
    );
```

要查询在cursor中的行，使用cursor的其中一个move方法，但必须在读取值之前调用。一般来说应该先调用 `moveToFirst()` 函数，将读取位置置于结果集最开始的位置。对每一行，我们可以使用cursor的其中一个get方法如 `getString()` 或 `getLong()` 获取列的值。对于每一个get方法必须传递想要获取的列的索引位置(index position)，索引位置可以通过调用 `getColumnIndex()` 或 `getColumnIndexOrThrow()` 获得。

下面演示如何从course对象中读取数据信息：

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedReaderContract.FeedEntry._ID)
);
```

删除DB中的信息

和查询信息一样，删除数据同样需要提供一些删除标准。DB的API提供了一个防止SQL注入的机制来创建查询与删除标准。

> **SQL Injection**：(随着*B/S*模式应用开发的发展，使用这种模式编写应用程序的程序员也越来越多。但由于程序员的水平及经验也参差不齐，相当大一部分程序员在编写代码时没有对用户输入数据的合法性进行判断，使应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果，获得某些他想得知的数据，这就是所谓的*SQL Injection*，即*SQL*注入)

该机制把查询语句划分为选项条件与选项参数两部分。条件定义了查询的列的特征，参数用于测试是否符合前面的条款。由于处理的结果不同于通常的SQL语句，这样可以避免SQL注入问题。

```
// Define 'where' part of query.
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selelectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, mySelection, selectionArgs);
```

更新数据

当需要修改DB中的某些数据时，使用 update() 方法。

update结合了插入与删除的语法。

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

与其他应用的交互

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/intents/index.html

- 一个Android app通常都会有多个activities。 每个activity的界面都扮演者用户接口的角色，允许用户执行一些特定任务（例如查看地图或者是开始拍照等）。为了让用户能够从一个activity跳到另一个activity，我们的app必须使用Intent来定义自己的意图。当使用startActivity()的方法，且参数是intent时，系统会使用这个 Intent 来定义并启动合适的app组件。使用intents甚至还可以让app启动另一个app里面的activity。
- 一个 Intent 可以显式的指明需要启动的模块（用一个指定的Activity实例），也可以隐式的指明自己可以处理哪种类型的动作（比如拍一张照等）。
- 本章节将演示如何使用Intent 与其他app执行一些基本的交互。比如启动另外一个app，从其他app接受数据，以及使得我们的app能够响应从其他app中发出的intent等。

Lessons

- **Intent的发送(Sending the User to Another App)**

  演示如何创建一个隐式Intent唤起能够接收这个动作的App。

- **接收Activity返回的结果(Getting a Result from an Activity)**

  演示如何启动另外一个Activity并接收返回值。

- **Intent过滤(Allowing Other Apps to Start Your Activity)**

  演示如何通过定义隐式的Intent的过滤器来使我们的应用能够被其他应用唤起。

Intent的发送

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/intents/sending.html

Android中最重要的特征之一就是可以利用一个带有 `action` 的 `intent` 使当前app能够跳转到其他app。例如：如果我们的app有一个地址想要显示在地图上，我们并不需要在app里面创建一个activity用来显示地图，而是使用Intent来发出查看地址的请求。Android系统则会启动能够显示地图的程序来呈现该地址。

正如在1.1章节:建立你的第一个App(Building Your First App)中所说的，我们必须使用intent来在同一个app的两个activity之间进行切换。通常是定义一个显式（explicit）的intent，它指定了需要启动组件的类名。然而，当想要唤起不同的app来执行某个动作（比如查看地图），则必须使用隐式（implicit）的intent。

本课会介绍如何为特殊的动作创建一个implicit intent，并使用它来启动另一个app去执行intent中的action。

建立隐式的Intent

Implicit intents并不声明要启动组件的具体类名，而是声明一个需要执行的action。这个action指定了我们想做的事情，例如查看，编辑，发送或者是获取一些东西。Intents通常会在发送action的同时附带一些数据，例如你想要查看的地址或者是你想要发送的邮件信息。数据的具体类型取决于我们想要创建的Intent，比如Uri或其他规定的数据类型，或者甚至也可能根本不需要数据。

如果数据是一个Uri，会有一个简单的Intent() constructor 用于定义action与data。

例如，下面是一个带有指定电话号码的intent。

```
Uri number = Uri.parse("tel:5551234");
Intent callIntent = new Intent(Intent.ACTION_DIAL, number);
```

当app通过执行startActivity()启动这个intent时，Phone app会使用之前的电话号码来拨出这个电话。

下面是一些其他intent的例子：

- 查看地图:

```
// Map point based on address
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+Californ
ia");
// Or map point based on latitude/longitude
// Uri location = Uri.parse("geo:37.422219,-122.08364?z=14"); // z param is zoom level
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
```

- 查看网页:

```
Uri webpage = Uri.parse("http://www.android.com");
Intent webIntent = new Intent(Intent.ACTION_VIEW, webpage);
```

至于另外一些需要 `extra` 数据的implicit intent，我们可以使用 putExtra() 方法来添加那些数据。 默认的，系统会根据Uri数据类型来决定需要哪些合适的 `MIME type` 。如果我们没有在intent中包含一个Uri, 则通常需要使用 setType() 方法来指定intent附带的数据类型。设置MIME type 是为了指定应该接受这个intent的activity。例如：

- 发送一个带附件的email:

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
// The intent does not have a URI, so declare the "text/plain" MIME type
emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"jon@example.com"}); // recipie
nts
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
emailIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("content://path/to/email/attachmen
t"));
// You can also attach multiple items by passing an ArrayList of Uris
```

- 创建一个日历事件:

```
Intent calendarIntent = new Intent(Intent.ACTION_INSERT, Events.CONTENT_URI);
Calendar beginTime = Calendar.getInstance().set(2012, 0, 19, 7, 30);
Calendar endTime = Calendar.getInstance().set(2012, 0, 19, 10, 30);
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME, beginTime.getTimeInMi
llis());
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_END_TIME, endTime.getTimeInMillis
());
calendarIntent.putExtra(Events.TITLE, "Ninja class");
calendarIntent.putExtra(Events.EVENT_LOCATION, "Secret dojo");
```

> **Note:** 这个intent for Calendar的例子只使用于>=API Level 14。
>
> **Note:** 请尽可能的将Intent定义的更加确切。例如，如果想要使用ACTION_VIEW 的intent来显示一张图片，则还应该指定 MIME type 为 `image/*`.这样能够阻止其他能够 "查看" 其他数据类型的app（比如一个地图app) 被这个intent叫起。

验证是否有App去接收这个Intent

尽管Android系统会确保每一个确定的intent会被系统内置的app(such as the Phone, Email, or Calendar app)之一接收，但是我们还是应该在触发一个intent之前做验证是否有App接受这个intent的步骤。

> **Caution:** 如果触发了一个intent，而且没有任何一个app会去接收这个intent，则app会crash。

为了验证是否有合适的activity会响应这个intent，需要执行queryIntentActivities() 来获取到能够接收这个intent的所有activity的list。若返回的List非空，那么我们才可以安全的使用这个intent。例如：

```
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(intent, 0);
boolean isIntentSafe = activities.size() > 0;
```

如果 `isIntentSafe` 为 `true` , 那么至少有一个app可以响应这个intent。 `false` 则说明没有app可以handle这个intent。

> **Note:**我们必须在第一次使用之前做这个检查，若是不可行，则应该关闭这个功能。如果知道某个确切的app能够handle这个intent，我们也可以向用户提供下载该app的链接。(see how to link to your product on Google Play).

使用Intent启动Activity

当创建好了intent并且设置好了extra数据后，通过执行startActivity() 将intent发送到系统。若系统确定了多个activity可以handle这个intent,它会显示出一个dialog，让用户选择启动哪个app。如果系统发现只有一个app可以handle这个intent，则系统将直接启动该app。

```
startActivity(intent);
```



下面是一个演示了如何创建一个intent来查看地图的完整例子，首先验证有app可以handle这个intent,然后启动它。

```
// Build the intent
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+Californ
ia");
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);

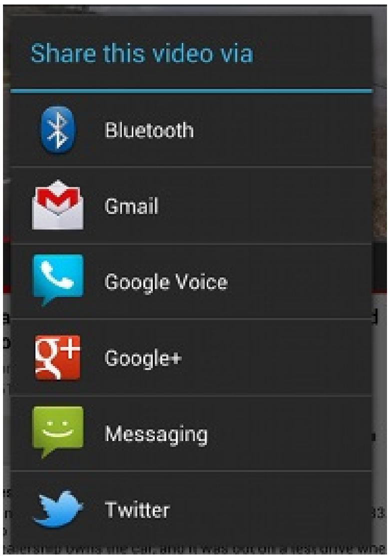
// Verify it resolves
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(mapIntent, 0);
boolean isIntentSafe = activities.size() > 0;

// Start an activity if it's safe
if (isIntentSafe) {
    startActivity(mapIntent);
}
```

显示分享App的选择界面

请注意，当以startActivity()的形式传递一个intent，并且有多个app可以handle时，用户可以在弹出dialog的时候选择默认启动的app（通过勾选dialog下面的选择框，如上图所示）。该功能对于用户有特殊偏好的时候非常有用（例如用户总是喜欢启动某个app来查看网页，总是喜欢启动某个camera来拍照）。

然而，如果用户希望每次都弹出选择界面，而且每次都不确定会选择哪个app启动，例如分享功能，用户选择分享到哪个app都是不确定的，这个时候，需要强制弹出选择的对话框。（这种情况下用户不能选择默认启动的app）。



为了显示chooser, 需要使用createChooser()来创建Intent

```
Intent intent = new Intent(Intent.ACTION_SEND);
...

// Always use string resources for UI text. This says something like "Share this photo
 with"
String title = getResources().getText(R.string.chooser_title);
// Create and start the chooser
Intent chooser = Intent.createChooser(intent, title);
startActivity(chooser);
```

这样就列出了可以响应 `createChooser()` 中Intent的app，并且指定了标题。

接收Activity返回的结果

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/intents/result.html

启动另外一个activity并不一定是单向的。我们也可以启动另外一个activity然后接受一个返回的result。为接受result，我们需要使用startActivityForResult() ，而不是startActivity()。

例如，我们的app可以启动一个camera程序并接受拍的照片作为result。或者可以启动联系人程序并获取其中联系的人的详情作为result。

当然，被启动的activity需要指定返回的result。它需要把这个result作为另外一个intent对象返回，我们的activity需要在onActivityResult()的回调方法里面去接收result。

> **Note:**在执行 `startActivityForResult()` 时，可以使用explicit 或者 implicit 的intent。当启动另外一个位于的程序中的activity时，我们应该使用explicit intent来确保可以接收到期待的结果。

启动Activity

对于startActivityForResult() 方法中的intent与之前介绍的并无太大差异，不过是需要在这个方法里面多添加一个int类型的参数。

该integer参数称为"request code"，用于标识请求。当我们接收到result Intent时，可从回调方法里面的参数去判断这个result是否是我们想要的。

例如，下面是一个启动activity来选择联系人的例子：

```
static final int PICK_CONTACT_REQUEST = 1;  // The request code
...
private void pickContact() {
    Intent pickContactIntent = new Intent(Intent.ACTION_PICK, Uri.parse("content://con
tacts"));
    pickContactIntent.setType(Phone.CONTENT_TYPE); // Show user only contacts w/ phone
 numbers
    startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST);
}
```

接收Result

当用户完成了启动之后activity操作之后，系统会调用我们activity中的onActivityResult() 回调方法。该方法有三个参数：

- 通过startActivityForResult()传递的request code。
- 第二个activity指定的result code。如果操作成功则是 `RESULT_OK` ，如果用户没有操作成功，而是直接点击回退或者其他什么原因，那么则是 `RESULT_CANCELED`
- 包含了所返回result数据的intent。

例如，下面显示了如何处理pick a contact的result：

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // Check which request we're responding to
    if (requestCode == PICK_CONTACT_REQUEST) {
        // Make sure the request was successful
        if (resultCode == RESULT_OK) {
            // The user picked a contact.
            // The Intent's data Uri identifies which contact was selected.

            // Do something with the contact here (bigger example below)
        }
    }
}
```

本例中被返回的Intent使用Uri的形式来表示返回的联系人。

为正确处理这些result，我们必须了解那些result intent的格式。对于自己程序里面的返回result是比较简单的。Apps都会有一些自己的api来指定特定的数据。例如，People app (Contacts app on some older versions) 总是返回一个URI来指定选择的contact，Camera app 则是在 `data` 数据区返回一个 Bitmap （see the class about Capturing Photos).

读取联系人数据

上面的代码展示了如何获取联系人的返回结果，但没有说清楚如何从结果中读取数据，因为这需要更多关于content providers的知识。但如果想知道的话，下面是一段代码，展示如何从被选的联系人中读出电话号码。

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // Check which request it is that we're responding to
    if (requestCode == PICK_CONTACT_REQUEST) {
        // Make sure the request was successful
        if (resultCode == RESULT_OK) {
            // Get the URI that points to the selected contact
            Uri contactUri = data.getData();
            // We only need the NUMBER column, because there will be only one row in t
he result
            String[] projection = {Phone.NUMBER};

            // Perform the query on the contact to get the NUMBER column
            // We don't need a selection or sort order (there's only one result for th
e given URI)
            // CAUTION: The query() method should be called from a separate thread to
avoid blocking
            // your app's UI thread. (For simplicity of the sample, this code doesn't
do that.)
            // Consider using CursorLoader to perform the query.
            Cursor cursor = getContentResolver()
                    .query(contactUri, projection, null, null, null);
            cursor.moveToFirst();

            // Retrieve the phone number from the NUMBER column
            int column = cursor.getColumnIndex(Phone.NUMBER);
            String number = cursor.getString(column);

            // Do something with the phone number...
        }
    }
}
```

> **Note**:在Android 2.3 (API level 9)之前对 `Contacts Provider` 的请求(比如上面的代码)，需要声明 `READ_CONTACTS` 权限(更多详见Security and Permissions)。但如果是Android 2.3以上的系统就不需要这么做。但这种临时权限也仅限于特定的请求，所以仍无法获取除返回的Intent以外的联系人信息，除非声明了 `READ_CONTACTS` 权限。

Intent过滤

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/intents/filters.html

前两节课主要讲了从一个app启动另外一个app。但如果我们的app的功能对别的app也有用，那么其应该做好响应的准备。例如，如果创建了一个social app，它可以分享messages 或者photos 给好友，那么最好我们的app能够接收 `ACTION_SEND` 的intent,这样当用户在其他app触发分享功能的时候，我们的app能够出现在待选对话框。

通过在manifest文件中的 `<activity>` 标签下添加 `<intent-filter>` 的属性，使其他的app能够启动我们的activity。

当app被安装到设备上时，系统可以识别intent filter并把这些信息记录下来。当其他app使用implicit intent执行 startActivity() 或者 startActivityForResult()时，系统会自动查找出那些可以响应该intent的activity。

添加Intent Filter

为了尽可能确切的定义activity能够handle的intent，每一个intent filter都应该尽可能详尽的定义好action与data。

若activity中的intent filter满足以下intent对象的标准，系统就能够把特定的intent发送给activity：

- **Action**:一个想要执行的动作的名称。通常是系统已经定义好的值，如 `ACTION_SEND` 或 `ACTION_VIEW` 。 在intent filter中通过 `<action>` 指定它的值，值的类型必须为字符串，而不是API中的常量(看下面的例子)

- **Data**:Intent附带数据的描述。在intent filter中通过 `<data>` 指定它的值，可以使用一个或者多个属性，我们可以只定义MIME type或者是只指定URI prefix，也可以只定义一个URI scheme，或者是他们综合使用。

> **Note:** 如果不想handle Uri 类型的数据，那么应该指定 android:mimeType 属性。例如text/plain or image/jpeg.

- **Category**:提供一个附加的方法来标识这个activity能够handle的intent。通常与用户的手势或者是启动位置有关。系统有支持几种不同的categories,但是大多数都很少用到。而且，所有的implicit intents都默认是 CATEGORY_DEFAULT 类型的。在intent filter中用 `<category>` 指定它的值。

在我们的intent filter中，可以在 `<intent-filter>` 元素中定义对应的XML元素来声明我们的activity使用何种标准。

例如，这个有intent filter的activity，当数据类型为文本或图像时会处理 `ACTION_SEND` 的intent。

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
        <data android:mimeType="image/*"/>
    </intent-filter>
</activity>
```

每一个发送出来的intent只会包含一个action与data类型，但handle这个intent的activity的 `<intent-filter>` 可以声明多个 `<action>`, `<category>` 与 `<data>` 。

如果任何的两对action与data是互相矛盾的，就应该创建不同的intent filter来指定特定的action与type。

例如，假设我们的activity可以handle 文本与图片，无论是 `ACTION_SEND` 还是 `ACTION_SENDTO` 的intent。在这种情况下，就必须为两个action定义两个不同的intent filter。因为 `ACTION_SENDTO` intent 必须使用 Uri 类型来指定接收者使用 send 或 sendto 的地址。例如：

```
<activity android:name="ShareActivity">
    <!-- filter for sending text; accepts SENDTO action with sms URI schemes -->
    <intent-filter>
        <action android:name="android.intent.action.SENDTO"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
    </intent-filter>
    <!-- filter for sending text or images; accepts SEND action and text or image data
 -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

> **Note:**为了接受implicit intents, 必须在我们的intent filter中包含 CATEGORY_DEFAULT 的category。startActivity()和startActivityForResult()方法将所有intent视为声明了CATEGORY_DEFAULT category。如果没有在的intent filter中声明CATEGORY_DEFAULT，activity将无法对implicit intent做出响应。

更多sending 与 receiving ACTION_SEND intents执行social sharing行为的，请查看上一课：接收Activity返回的结果(Getting a Result from an Activity)

在Activity中Handle发送过来的Intent

为了决定采用哪个action，我们可以读取Intent的内容。

可以执行getIntent() 来获取启动我们activity的那个intent。我们可以在activity生命周期的任何时候去执行这个方法，但最好是在 `onCreate()` 或者 `onStart()` 里面去执行。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

    // Get the intent that started this activity
    Intent intent = getIntent();
    Uri data = intent.getData();

    // Figure out what to do based on the intent type
    if (intent.getType().indexOf("image/") != -1) {
        // Handle intents with image data ...
    } else if (intent.getType().equals("text/plain")) {
        // Handle intents with text ...
    }
}
```

返回Result

如果想返回一个result给启动的那个activity，仅仅需要执行setResult()，通过指定一个result code与result intent。操作完成之后，用户需要返回到原来的activity，通过执行finish() 关闭被唤起的activity。

```
 // Create intent to deliver some kind of result data
Intent result = new Intent("com.example.RESULT_ACTION", Uri.parse("content://result_ur
i");
setResult(Activity.RESULT_OK, result);
finish();
```

我们必须总是指定一个result code。通常不是 `RESULT_OK` 就是 `RESULT_CANCELED` 。我们可以通过Intent 来添加需要返回的数据。

> **Note:**默认的result code是 `RESULT_CANCELED` .因此，如果用户在没有完成操作之前点击了back key，那么之前的activity接受到的result code就是"canceled"。

如果只是纯粹想要返回一个int来表示某些返回的result数据之一，则可以设置result code为任何大于0的数值。如果我们返回的result只是一个int，那么连intent都可以不需要返回了，可以调用 `setResult()` 然后只传递result code如下：

```
setResult(RESULT_COLOR_RED);
finish();
```

> **Note:**我们没有必要在意自己的activity是被用startActivity() 还是 startActivityForResult() 方法所叫起的。系统会自动去判断该如何传递result。在不需要的result的case下，result会被自动忽略。

Android分享操作(Building Apps with Content Sharing)

> 编写:kesenhoo - 原文:http://developer.android.com/training/building-content-sharing.html

这一系列课程会教你如何创建可以在不同的应用与设备之间进行分享的应用。

分享简单的数据(Sharing Simple Data)

学习如何使得你的应用可以和其他应用进行交互。分享信息，接收信息，为用户数据提供一个简单并且可扩展的方式来执行分享操作。

分享文件(Sharing Files)

学习使用一个URI与临时的访问权限来提供安全的文件访问。

使用NFC分享文件(Sharing Files with NFC)

学习使用NFC功能实现设备间的文件传递。

分享简单的数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/sharing/index.html

程序间可以互相通信是Android程序中最棒的功能之一。当一个功能已存在于其他app中，且并不是本程序的核心功能时，完全没有必要重新对其进行编写。

本章节会讲述一些通在不同程序之间通过使用Intent APIs与ActionProvider对象来发送与接受content的常用方法。

Lessons

- **向其他App发送简单的数据 - Sending Simple Data to Other Apps**

  学习如何使用intent向其他app发送text与binary数据。

- **接收从其他App返回的数据 - Receiving Simple Data from Other Apps**

  学习如何通过Intent在我们的app中接收来自其他app的text与binary数据。

- **给ActionBar增加分享功能 - Adding an Easy Share Action**

  学习如何在Acitonbar上添加一个分享功能。

给其他App发送简单的数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/sharing/send.html

在构建一个intent时，必须指定这个intent需要触发的actions。Android定义了一些actions，比如ACTION_SEND，该action表明该intent用于从一个activity发送数据到另外一个activity的，甚至可以是跨进程之间的数据发送。

为了发送数据到另外一个activity，我们只需要指定数据与数据的类型，系统会自动识别出能够兼容接受的这些数据的activity。如果这些选择有多个，则把这些activity显示给用户进行选择；如果只有一个，则立即启动该Activity。同样的，我们可以在manifest文件的Activity描述中添加接受的数据类型。

在不同的程序之间使用intent收发数据是在社交分享内容时最常用的方法。Intent使用户能够通过最常用的程序进行快速简单的分享信息。

注意:为ActionBar添加分享功能的最佳方法是使用ShareActionProvider，其运行与API level 14以上的系统。ShareActionProvider将在第3课中进行详细介绍。

分享文本内容(Send Text Content)

ACTION_SEND最直接常用的地方是从一个Activity发送文本内容到另外一个Activity。例如，Android内置的浏览器可以将当前显示页面的URL作为文本内容分享到其他程序。这一功能对于通过邮件或者社交网络来分享文章或者网址给好友而言是非常有用的。下面是一段Sample Code:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

如果设备上安装有某个能够匹配ACTION_SEND且MIME类型为text/plain的程序，则Android系统会立即执行它。若有多个匹配的程序，则系统会把他们都给筛选出来，并呈现Dialog给用户进行选择。

如果为intent调用了Intent.createChooser()，那么Android总是会显示可供选择。这样有一些好处：

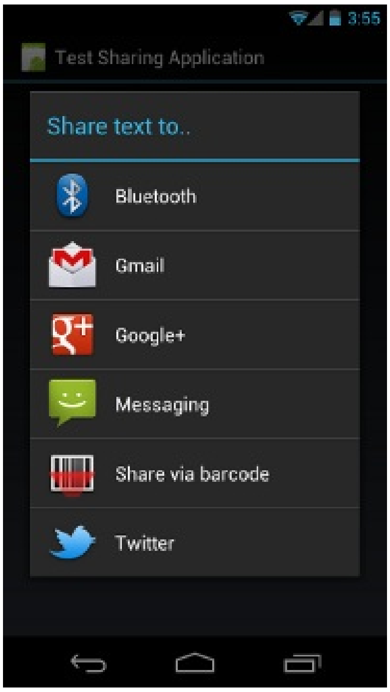
- 即使用户之前为这个intent设置了默认的action，选择界面还是会被显示。
- 如果没有匹配的程序，Android会显示系统信息。

- 我们可以指定选择界面的标题。

下面是更新后的代码：

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent, getResources().getText(R.string.send_to
));
```

效果图如下：



另外,我们可以为intent设置一些标准的附加值，例如：EXTRA_EMAIL, EXTRA_CC, EXTRA_BCC, EXTRA_SUBJECT等。然而，如果接收程序没有针对那些做特殊的处理，则不会有对应的反应。

注意:一些e-mail程序，例如Gmail,对应接收的是EXTRA_EMAIL与EXTRA_CC，他们都是String类型的，可以使用putExtra(string,string[])方法来添加至intent中。

分享二进制内容(Send Binary Content)

分享二进制的数据需要结合设置特定的MIME类型 `，需要在` EXTRA_STREAM`里面放置数据的URI,下面有个分享图片的例子，该例子也可以修改用于分享任何类型的二进制数据：

```
Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND);
shareIntent.putExtra(Intent.EXTRA_STREAM, uriToImage);
shareIntent.setType("image/jpeg");
startActivity(Intent.createChooser(shareIntent, getResources().getText(R.string.send_t
o)));
```

请注意以下内容：

- 我们可以使用 `*/*` 这样的方式来指定MIME类型，但是这仅仅会match到那些能够处理一般数据类型的Activity(即一般的Activity无法详尽所有的MIME类型)
- 接收的程序需要有访问URI资源的权限。下面有一些方法来处理这个问题：
  - 将数据存储在ContentProvider中，确保其他程序有访问provider的权限。较好的提供访问权限的方法是使用 per-URI permissions，其对接收程序而言是只是暂时拥有该许可权限。类似于这样创建ContentProvider的一种简单的方法是使用FileProvider helper类。
  - 使用MediaStore系统。MediaStore系统主要用于音视频及图片的MIME类型。但在Android3.0之后，其也可以用于存储非多媒体类型。

发送多块内容(Send Multiple Pieces of Content)

为了同时分享多种不同类型的内容，需要使用 `ACTION_SEND_MULTIPLE` 与指定到那些数据的URIs列表。MIME类型会根据分享的混合内容而不同。例如，如果分享3张JPEG的图片，那么MIME类型仍然是 `image/jpeg` 。如果是不同图片格式的话，应该是用 `image/*` 来匹配那些可以接收任何图片类型的activity。如果需要分享多种不同类型的数据，可以使用 `*/*` 来表示MIME。像前面描述的那样，这取决于那些接收的程序解析并处理我们的数据。下面是一个例子：

```
ArrayList<Uri> imageUris = new ArrayList<Uri>();
imageUris.add(imageUri1); // Add your image URIs here
imageUris.add(imageUri2);

Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUris);
shareIntent.setType("image/*");
startActivity(Intent.createChooser(shareIntent, "Share images to.."));
```

当然，请确保指定到数据的URIs能够被接收程序所访问(添加访问权限)。

接收从其他App传送来的数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/sharing/receive.html

就像我们的程序能够分享数据给其他程序一样，其也能方便的接收来自其他程序的数据。需要考虑的是用户与我们的程序如何进行交互，以及我们想要从其他程序接收数据的类型。例如，一个社交网络程序可能会希望能够从其他程序接受文本数据，比如一个有趣的网址链接。Google+的Android客户端会接受文本数据与单张或者多张图片。用户可以简单的从Gallery程序选择一张图片来启动Google+，并利用其发布文本或图片。

更新我们的manifest文件(Update Your Manifest)

Intent filters告诉Android系统一个程序愿意接受的数据类型。类似于上一课，我们可以创建intent filters来表明程序能够接收的action类型。下面是个例子，对三个activit分别指定接受单张图片，文本与多张图片。(Intent filter相关资料，请参考Intents and Intent Filters)

```
<activity android:name=".ui.MyActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND_MULTIPLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
</activity>
```

当某个程序尝试通过创建一个intent并将其传递给startActivity来分享一些东西时，我们的程序会被呈现在一个列表中让用户进行选择。如果用户选择了我们的程序，相应的activity会被调用开启，这个时候就是我们如何处理获取到的数据的问题了。

处理接受到的数据(Handle the Incoming Content)

为了处理从Intent带来的数据，可以通过调用getIntent()方法来获取到Intent对象。拿到这个对象后，我们可以对其中面的数据进行判断，从而决定下一步行为。请记住，如果一个activity可以被其他的程序启动，我们需要在检查intent的时候考虑这种情况(是被其他程序而调用启动的)。

```
void onCreate (Bundle savedInstanceState) {
    ...
    // Get intent, action and MIME type
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
        if ("text/plain".equals(type)) {
            handleSendText(intent); // Handle text being sent
        } else if (type.startsWith("image/")) {
            handleSendImage(intent); // Handle single image being sent
        }
    } else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type != null) {
        if (type.startsWith("image/")) {
            handleSendMultipleImages(intent); // Handle multiple images being sent
        }
    } else {
        // Handle other intents, such as being started from the home screen
    }
    ...
}

void handleSendText(Intent intent) {
    String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
    if (sharedText != null) {
        // Update UI to reflect text being shared
    }
}

void handleSendImage(Intent intent) {
    Uri imageUri = (Uri) intent.getParcelableExtra(Intent.EXTRA_STREAM);
    if (imageUri != null) {
        // Update UI to reflect image being shared
    }
}

void handleSendMultipleImages(Intent intent) {
    ArrayList<Uri> imageUris = intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM)
;
    if (imageUris != null) {
        // Update UI to reflect multiple images being shared
    }
}
```

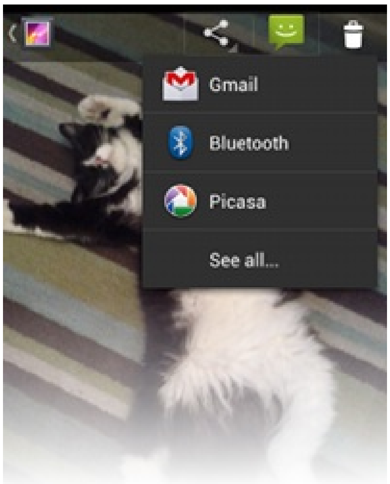
请注意，由于无法知道其他程序发送过来的数据内容是文本还是其他类型的数据，若数据量巨大，则需要大量处理时间，因此我们应避免在UI线程里面去处理那些获取到的数据。

更新UI可以像更新EditText一样简单，也可以是更加复杂一点的操作，例如过滤出感兴趣的图片。这完全取决于我们的应用接下来要做些什么。

添加一个简便的分享功能

> 编写:kesenhoo - 原文:http://developer.android.com/training/sharing/shareaction.html

Android4.0之后系统中ActionProvider的引入使在ActionBar中添加分享功能变得更为简单。它会handle出现share功能的appearance与behavior。在ShareActionProvider的例子里面，我们只需要提供一个share intent，剩下的就交给ShareActionProvider来做。



更新菜单声明(Update Menu Declarations)

使用ShareActionProvider的第一步，在menu resources对应item中定义 `android:actionProviderClass` 属性。

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_item_share"
        android:showAsAction="ifRoom"
        android:title="Share"
        android:actionProviderClass="android.widget.ShareActionProvider" />
    ...
</menu>
```

这表明了该item的appearance与function需要与ShareActionProvider匹配。此外，你还需要告诉provider想分享的内容。

Set the Share Intent(设置分享的intent)

为了实现ShareActionProvider的功能，我们必须为它提供一个intent。该share intent应该像第一课讲的那样，带有 `ACTION_SEND` 和附加数据(例如 `EXTRA_TEXT` 与 `EXTRA_STREAM`)的。使用ShareActionProvider的例子如下：

```
private ShareActionProvider mShareActionProvider;
...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate menu resource file.
    getMenuInflater().inflate(R.menu.share_menu, menu);

    // Locate MenuItem with ShareActionProvider
    MenuItem item = menu.findItem(R.id.menu_item_share);

    // Fetch and store ShareActionProvider
    mShareActionProvider = (ShareActionProvider) item.getActionProvider();

    // Return true to display menu
    return true;
}

// Call to update the share intent
private void setShareIntent(Intent shareIntent) {
    if (mShareActionProvider != null) {
        mShareActionProvider.setShareIntent(shareIntent);
    }
}
```

也许在创建菜单的时候仅仅需要设置一次share intent就满足需求了，或者说我们可能想先设置share intent，然后根据UI的变化来对intent进行更新。例如，当在Gallery里面全图查看照片的时候，share intent会在切换图片时候进行改变。 更多关于ShareActionProvider的内容，请查看Action Bar 。

分享文件

> 编写:jdneo - 原文:http://developer.android.com/training/secure-file-sharing/index.html

一个程序经常需要向其他程序提供一个甚至多个文件。例如，当我们用图片编辑器编辑图片时，被编辑的图片往往由图库应用程序所提供；再比如，文件管理器会允许用户在外部存储的不同区域之间复制粘贴文件。这里，我们提出一种让应用程序可以分享文件的方法：即令发送文件的应用程序对索取文件的应用程序所发出的文件请求进行响应。

在任何情况下，将文件从我们的应用程序发送至其它应用程序的唯一的安全方法是向接收文件的应用程序发送这个文件的content URI，并对该URI授予临时访问权限。具有URI临时访问权限的content URI是安全的，因为他们仅应用于接收这个URI的应用程序，并且会自动过期。Android的FileProvider组件提供了getUriForFile()方法创建一个文件的content URI。

如果希望在应用之间共享少量的文本或者数字等类型的数据，应使用包含该数据的Intent。要学习如何通过Intent发送简单数据，可以阅读：Sharing Simple Data。

本课主要介绍了如何使用Android的FileProvider组件所创建的content URI在应用之间安全的共享文件。当然，要做到这一点，还需要给接收文件的应用程序访问的这些content URI授予临时访问权限。

Lessons

- 建立文件分享

  学习如何配置应用程序使得它们可以分享文件。

- 分享文件

  学习分享文件的三个步骤：

  - 生成文件的content URI；
  - 授予URI的临时访问权限；
  - 将URI发送给接收文件的应用程序。

- 请求分享一个文件

  学习如何向其他应用程序请求文件，如何接收该文件的content URI，以及如何使用content URI打开该文件。

- 获取文件信息

学习应用程序如何通过FileProvider提供的content URI获取文件的信息：例如MIME类型，文件大小等。

建立文件分享

> 编写:jdneo - 原文:http://developer.android.com/training/secure-file-sharing/setup-sharing.html

为了将文件安全地从我们的应用程序共享给其它应用程序，我们需要对自己的应用进行配置来提供安全的文件句柄（Content URI的形式）。Android的FileProvider组件会基于在XML文件中的具体配置为文件创建Content URI。本课将介绍如何在应用程序中添加FileProvider的默认实现，以及如何指定要共享的文件。

> **Note:**FileProvider类隶属于v4 Support Library库。关于如何在应用程序中包含该库，请参考：Support Library Setup。

指定FileProvider

为了给应用程序定义一个FileProvider，需要在Manifest清单文件中定义一个entry，该entry指明了需要使用的创建Content URI的Authority。此外，还需要一个XML文件的文件名，该XML文件指定了我们的应用可以共享的目录路径。

下例展示了如何在清单文件中添加 `<provider>` 标签，来指定FileProvider类，Authority及XML文件名：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application
        ...>
        <provider
            android:name="android.support.v4.content.FileProvider"
            android:authorities="com.example.myapp.fileprovider"
            android:grantUriPermissions="true"
            android:exported="false">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/filepaths" />
        </provider>
        ...
    </application>
</manifest>
```

这里，android:authorities字段指定了希望使用的Authority，该Authority针对于FileProvider所生成的content URI。本例中的Authority是“com.example.myapp.fileprovider”。对于自己的应用，要在我们的应用程序包名（android:package的值）之后继续追加“fileprovider”来指定

Authority。要更多关于Authority的知识，请参考：Content URIs，以及android:authorities。

`<provider>` 下的 `<meta-data>` 指向了一个XML文件，该文件指定了我们希望共享的目录路径。“android:resource”属性字段是这个文件的路径和名字（无“.xml”后缀）。该文件的内容将在下一节讨论。

指定可共享目录路径

一旦在Manifest清单文件中为自己的应用添加了FileProvider，就需要指定我们希望共享文件的目录路径。为指定该路径，首先要在“res/xml/”下创建文件“filepaths.xml”。在这个文件中，为每一个想要共享目录添加一个XML标签。下面的是一个“res/xml/filepaths.xml”的内容样例。这个例子也说明了如何在内部存储区域共享一个“files/”目录的子目录：

```
<paths>
    <files-path path="images/" name="myimages" />
</paths>
```

在这个例子中， `<files-path>` 标签共享的是在我们应用的内部存储中“files/”目录下的目录。“path”属性字段指出了该子目录为“files/”目录下的子目录“images/”。“name”属性字段告知FileProvider在“files/images/”子目录中的文件的Content URI添加路径分段（path segment）标记：“myimages”。

`<paths>` 标签可以有多个子标签，每一个子标签用来指定不同的共享目录。除了 `<files-path>` 标签，还可以使用 `<external-path>` 来共享位于外部存储的目录；另外， `<cache-path>` 标签用来共享在内部缓存目录下的子目录。更多关于指定共享目录子标签的知识请参考：FileProvider。

> **Note:** XML文件是我们定义共享目录的唯一方式，不可以用代码的形式添加目录。

现在我们有一个完整的FileProvider声明，它在应用程序的内部存储中“files/”目录或其子目录下创建文件的Content URI。当我们的应用为一个文件创建了Content URI，该Content URI将会包含下列信息：

- `<provider>` 标签中指定的Authority（“com.example.myapp.fileprovider”）；
- 路径“myimages/”；
- 文件的名字。

例如，如果本课的例子定义了一个FileProvider，然后我们需要一个文件“default_image.jpg”的Content URI，FileProvider会返回如下URI：

```
content://com.example.myapp.fileprovider/myimages/default_image.jpg
```

分享文件

> 编写:jdneo - 原文:http://developer.android.com/training/secure-file-sharing/sharing-file.html

对应用程序进行配置，使得它可以使用Content URI来共享文件后，其就可以响应其他应用程序的获取文件的请求了。一种响应这些请求的方法是在服务端应用程序提供一个可以由其他应用激活的文件选择接口。该方法可以允许客户端应用程序让用户从服务端应用程序选择一个文件，然后接收这个文件的Content URI。

本课将会展示如何在应用中创建一个用于选择文件的Activity，来响应这些获取文件的请求。

接收文件请求

为了从客户端应用程序接收一个文件获取请求并以Content URI的形式进行响应，我们的应用程序应该提供一个选择文件的Activity。客户端应用程序通过调用startActivityForResult()方法启动这一Activity。该方法包含了一个具有ACTION_PICKAction的Intent参数。当客户端应用程序调用了startActivityForResult()，我们的应用可以向客户端应用程序返回一个结果，该结果即用户所选择的文件所对应的Content URI。

关于如何在客户端应用程序实现文件获取请求，请参考：请求分享一个文件。

创建一个选择文件的Activity

为建立一个选择文件的Activity，首先需要在Manifest清单文件中定义Activity，在其Intent过滤器中，匹配ACTION_PICKAction及CATEGORY_DEFAULT和CATEGORY_OPENABLE这两种Category。另外，还需要为应用程序设置MIME类型过滤器，来表明我们的应用程序可以向其他应用程序提供哪种类型的文件。下面这段代码展示了如何在清单文件中定义新的Activity和Intent过滤器：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
        <application>
        ...
            <activity
                android:name=".FileSelectActivity"
                android:label="File Selector" >
                <intent-filter>
                    <action
                        android:name="android.intent.action.PICK"/>
                    <category
                        android:name="android.intent.category.DEFAULT"/>
                    <category
                        android:name="android.intent.category.OPENABLE"/>
                    <data android:mimeType="text/plain"/>
                    <data android:mimeType="image/*"/>
                </intent-filter>
            </activity>
```

在代码中定义文件选择Activity

下面，定义一个Activity子类，用于显示在内部存储的“files/images/”目录下可以获得的文件，然后允许用户选择期望的文件。下面代码展示了如何定义该Activity，并令其响应用户的选择：

```
public class MainActivity extends Activity {
    // The path to the root of this app's internal storage
    private File mPrivateRootDir;
    // The path to the "images" subdirectory
    private File mImagesDir;
    // Array of files in the images subdirectory
    File[] mImageFiles;
    // Array of filenames corresponding to mImageFiles
    String[] mImageFilenames;
    // Initialize the Activity
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Set up an Intent to send back to apps that request a file
        mResultIntent =
                new Intent("com.example.myapp.ACTION_RETURN_FILE");
        // Get the files/ subdirectory of internal storage
        mPrivateRootDir = getFilesDir();
        // Get the files/images subdirectory;
        mImagesDir = new File(mPrivateRootDir, "images");
        // Get the files in the images subdirectory
        mImageFiles = mImagesDir.listFiles();
        // Set the Activity's result to null to begin with
        setResult(Activity.RESULT_CANCELED, null);
        /*
         * Display the file names in the ListView mFileListView.
         * Back the ListView with the array mImageFilenames, which
         * you can create by iterating through mImageFiles and
         * calling File.getAbsolutePath() for each File
         */
         ...
    }
    ...
}
```

响应一个文件选择

一旦用户选择了一个被共享的文件，我们的应用程序必须明确哪个文件被选择了，并为该文件生成一个对应的Content URI。如果我们的Activity在ListView中显示了可获得文件的清单，那么当用户点击了一个文件名时，系统会调用方法onItemClick()，在该方法中可以获取被选择的文件。

在onItemClick()中，根据被选中文件的文件名获取一个File对象，然后将其作为参数传递给getUriForFile()，另外还需传入的参数是在 `<provider>` 标签中为FileProvider所指定的Authority，函数返回的Content URI包含了相应的Authority，一个对应于文件目录的路径标记（如在XML meta-data中定义的），以及包含扩展名的文件名。有关FileProvider如何基于XML meta-data将目录路径与路径标记进行匹配的知识，可以阅读：指定可共享目录路径。

下例展示了如何检测选中的文件并且获得它的Content URI：

```
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Define a listener that responds to clicks on a file in the ListView
        mFileListView.setOnItemClickListener(
                new AdapterView.OnItemClickListener() {
            @Override
            /*
             * When a filename in the ListView is clicked, get its
             * content URI and send it to the requesting app
             */
            public void onItemClick(AdapterView<?> adapterView,
                    View view,
                    int position,
                    long rowId) {
                /*
                 * Get a File for the selected file name.
                 * Assume that the file names are in the
                 * mImageFilename array.
                 */
                File requestFile = new File(mImageFilename[position]);
                /*
                 * Most file-related method calls need to be in
                 * try-catch blocks.
                 */
                // Use the FileProvider to get a content URI
                try {
                    fileUri = FileProvider.getUriForFile(
                            MainActivity.this,
                            "com.example.myapp.fileprovider",
                            requestFile);
                } catch (IllegalArgumentException e) {
                    Log.e("File Selector",
                          "The selected file can't be shared: " +
                          clickedFilename);
                }
                ...
            }
        });
        ...
    }
```

记住，我们能生成的那些Content URI所对应的文件，必须是那些在meta-data文件中包含 `<paths>` 标签的目录内的文件，这方面知识在Specify Sharable Directories中已经讨论过。如果调用getUriForFile()方法所要获取的文件不在我们指定的目录中，会收到一个IllegalArgumentException。

为文件授权

现在已经有了想要共享给其他应用程序的文件所对应的Content URI，我们需要允许客户端应用程序访问这个文件。为了达到这一目的，可以通过将Content URI添加至一个Intent中，然后为该Intent设置权限标记。所授予的权限是临时的，并且当接收文件的应用程序的任务栈终止后，会自动过期。

下例展示了如何为文件设置读权限：

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Define a listener that responds to clicks in the ListView
    mFileListView.setOnItemClickListener(
            new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView,
                View view,
                int position,
                long rowId) {
            ...
            if (fileUri != null) {
                // Grant temporary read permission to the content URI
                mResultIntent.addFlags(
                    Intent.FLAG_GRANT_READ_URI_PERMISSION);
            }
            ...
         }
         ...
    });
...
}
```

> **Caution**：调用setFlags()来为文件授予临时被访问权限是唯一的安全的方法。尽量避免对文件的Content URI调用Context.grantUriPermission()，因为通过该方法授予的权限，只能通过调用Context.revokeUriPermission()来撤销。

与请求应用共享文件

为了向请求文件的应用程序提供其需要的文件，我们将包含了Content URI和相应权限的Intent传递给setResult()。当定义的Activity结束后，系统会把这个包含了Content URI的Intent传递给客户端应用程序。下例展示了其中的核心步骤：

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Define a listener that responds to clicks on a file in the ListView
    mFileListView.setOnItemClickListener(
            new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView,
                View view,
                int position,
                long rowId) {
            ...
            if (fileUri != null) {
                ...
                // Put the Uri and MIME type in the result Intent
                mResultIntent.setDataAndType(
                        fileUri,
                        getContentResolver().getType(fileUri));
                // Set the result
                MainActivity.this.setResult(Activity.RESULT_OK,
                        mResultIntent);
                } else {
                    mResultIntent.setDataAndType(null, "");
                    MainActivity.this.setResult(RESULT_CANCELED,
                            mResultIntent);
                }
            }
    });
```

当用户选择好文件后，我们应该向用户提供一个能够立即回到客户端应用程序的方法。一种实现的方法是向用户提供一个勾选框或者一个完成按钮。可以使用按钮的android:onClick属性字段为它关联一个方法。在该方法中，调用finish()。例如：

```
public void onDoneClick(View v) {
    // Associate a method with the Done button
    finish();
}
```

请求分享一个文件

> 编写:jdneo - 原文:http://developer.android.com/training/secure-file-sharing/request-file.html

当一个应用程序希望访问由其它应用程序所共享的文件时，请求应用程序（客户端）经常会向其它应用程序（服务端）发送一个文件请求。多数情况下，该请求会导致在服务端应用程序中启动一个Activity，该Activity中会显示可以共享的文件。当服务端应用程序向客户端应用程序返回了文件的Content URI后，用户即可开始选择文件。

本课将展示一个客户端应用程序应该如何向服务端应用程序请求一个文件，接收服务端应用程序发来的Content URI，然后使用这个Content URI打开这个文件。

发送一个文件请求

为了向服务端应用程序发送文件请求，在客户端应用程序中，需要调用startActivityForResult)方法，同时传递给这个方法一个Intent参数，它包含了客户端应用程序能处理的某个Action，比如ACTION_PICK及一个MIME类型。

例如，下面的代码展示了如何向服务端应用程序发送一个Intent，来启动在分享文件中提到的Activity：

```
public class MainActivity extends Activity {
    private Intent mRequestFileIntent;
    private ParcelFileDescriptor mInputPFD;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mRequestFileIntent = new Intent(Intent.ACTION_PICK);
        mRequestFileIntent.setType("image/jpg");
        ...
    }
    ...
    protected void requestFile() {
        /**
         * When the user requests a file, send an Intent to the
         * server app.
         * files.
         */
            startActivityForResult(mRequestFileIntent, 0);
        ...
    }
    ...
}
```

访问请求的文件

当服务端应用程序向客户端应用程序发回包含Content URI的Intent时，该Intent会传递给客户端应用程序重写的onActivityResult()方法当中。一旦客户端应用程序拥有了文件的Content URI，它就可以通过获取其FileDescriptor访问文件了。

这一过程中不用过多担心文件的安全问题，因为客户端应用程序所收到的所有数据只有文件的Content URI而已。由于URI不包含目录路径信息，客户端应用程序无法查询或打开任何服务端应用程序的其他文件。客户端应用程序仅仅获取了这个文件的访问渠道以及由服务端应用程序授予的访问权限。同时访问权限是临时的，一旦这个客户端应用的任务栈结束了，这个文件将无法再被除服务端应用程序之外的其他应用程序访问。

下面的例子展示了客户端应用程序应该如何处理发自服务端应用程序的Intent，以及客户端应用程序如何使用Content URI获取FileDescriptor：

```
    /*
     * When the Activity of the app that hosts files sets a result and calls
     * finish(), this method is invoked. The returned Intent contains the
     * content URI of a selected file. The result code indicates if the
     * selection worked or not.
     */
    @Override
    public void onActivityResult(int requestCode, int resultCode,
            Intent returnIntent) {
        // If the selection didn't work
        if (resultCode != RESULT_OK) {
            // Exit without doing anything else
            return;
        } else {
            // Get the file's content URI from the incoming Intent
            Uri returnUri = returnIntent.getData();
            /*
             * Try to open the file for "read" access using the
             * returned URI. If the file isn't found, write to the
             * error log and return.
             */
            try {
                /*
                 * Get the content resolver instance for this context, and use it
                 * to get a ParcelFileDescriptor for the file.
                 */
                mInputPFD = getContentResolver().openFileDescriptor(returnUri, "r");
            } catch (FileNotFoundException e) {
                e.printStackTrace();
                Log.e("MainActivity", "File not found.");
                return;
            }
            // Get a regular file descriptor for the file
            FileDescriptor fd = mInputPFD.getFileDescriptor();
            ...
        }
    }
```

openFileDescriptor()方法返回一个文件的ParcelFileDescriptor对象。客户端应用程序从该对象中获取FileDescriptor对象，然后利用该对象读取这个文件了。

获取文件信息

> 编写:jdneo - 原文:http://developer.android.com/training/secure-file-sharing/retrieve-info.html

当一个客户端应用程序拥有了文件的Content URI之后，它就可以获取该文件并进行下一步的工作了，但在此之前，客户端应用程序还可以向服务端应用程序获取关于文件的信息，包括文件的数据类型和文件大小等等。数据类型可以帮助客户端应用程序确定自己能否处理该文件，文件大小能帮助客户端应用程序为文件设置合理的缓冲区。

本课将展示如何通过查询服务端应用程序的FileProvider来获取文件的MIME类型和文件大小。

获取文件的MIME类型

客户端应用程序可以通过文件的数据类型判断自己应该如何处理这个文件的内容。客户端应用程序可以通过调用ContentResolver.getType()方法获得Content URI所对应的文件数据类型。该方法返回文件的MIME类型。默认情况下，一个FileProvider通过文件的后缀名来确定其MIME类型。

下例展示了当服务端应用程序将Content URI返回给客户端应用程序后，客户端应用程序应该如何获取文件的MIMIE类型：

```
...
/*
 * Get the file's content URI from the incoming Intent, then
 * get the file's MIME type
 */
Uri returnUri = returnIntent.getData();
String mimeType = getContentResolver().getType(returnUri);
...
```

获取文件名及文件大小

FileProvider类有一个query()方法的默认实现，它返回一个Cursor对象，该Cursor对象包含了Content URI所关联的文件的名称和大小。默认的实现返回下面两列信息：

DISPLAY_NAME

文件名，String类型。这个值和File.getName()所返回的值一样。

SIZE

文件大小，以字节为单位，long类型。这个值和File.length()所返回的值一样。

客户端应用可以通过将query()的除了Content URI之外的其他参数都设置为“null”，来同时获取文件的名称（DISPLAY_NAME）和大小（SIZE）。例如，下面的代码获取一个文件的名称和大小，然后在两个TextView中将他们显示出来：

```
...
/*
 * Get the file's content URI from the incoming Intent,
 * then query the server app to get the file's display name
 * and size.
 */
Uri returnUri = returnIntent.getData();
Cursor returnCursor =
        getContentResolver().query(returnUri, null, null, null, null);
/*
 * Get the column indexes of the data in the Cursor,
 * move to the first row in the Cursor, get the data,
 * and display it.
 */
int nameIndex = returnCursor.getColumnIndex(OpenableColumns.DISPLAY_NAME);
int sizeIndex = returnCursor.getColumnIndex(OpenableColumns.SIZE);
returnCursor.moveToFirst();
TextView nameView = (TextView) findViewById(R.id.filename_text);
TextView sizeView = (TextView) findViewById(R.id.filesize_text);
nameView.setText(returnCursor.getString(nameIndex));
sizeView.setText(Long.toString(returnCursor.getLong(sizeIndex)));
...
```

使用NFC分享文件

> 编写:jdneo - 原文:http://developer.android.com/training/beam-files/index.html

Android允许我们通过Android Beam文件传输功能在设备之间传送大文件。该功能具有简单的API，它使得用户仅需要通过一些简单的触控操作就能启动文件传输过程。Android Beam会自动地将文件从一台设备拷贝至另一台设备中，并在完成时告知用户。

Android Beam文件传输API可以用来处理规模较大的数据，而在Android4.0（API Level 14）引入的Android Beam NDEF传输API则用来处理规模较小的数据，如URI或者消息数据等。另外，Android Beam仅仅只是Android NFC框架提供的众多特性之一，它允许我们从NFC标签中读取NDEF消息。更多有关Android Beam的知识，请参考：Beaming NDEF Messages to Other Devices。更多有关NFC框架的知识，请参考：Near Field Communication。

Lessons

- 发送文件给其他设备

  学习如何配置应用程序，使其可以发送文件给其他设备。

- 接收其他设备的文件

  学习如何配置应用程序，使其可以接收其他设备发送的文件。

发送文件给其他设备

> 编写:jdneo - 原文:http://developer.android.com/training/beam-files/sending-files.html

这节课将展示如何通过Android Beam文件传输向另一台设备发送大文件。要发送文件，首先应声明使用NFC和外部存储的权限，我们需要测试一下自己的设备是否支持NFC，这样才能够将文件的URI提供给Android Beam文件传输。

使用Android Beam文件传输功能必须满足以下要求：

1. Android Beam文件传输功能传输大文件必须在Android 4.1（API Level 16）及以上版本的Android系统中使用。
2. 希望传送的文件必须放置于外部存储。更多关于外部存储的知识，请参考：Using the External Storage。
3. 希望传送的文件必须是全局可读的。我们可以通过File.setReadable(true,false)来为文件设置相应的读权限。
4. 必须提供待传输文件的File URI。Android Beam文件传输无法处理由FileProvider.getUriForFile生成的Content URI。

在清单文件中声明

首先，编辑Manifest清单文件来声明应用程序所需要的权限和功能。

声明权限

为了允许应用程序使用Android Beam文件传输控制NFC从外部存储发送文件，必须在应用程序的Manifest清单文件中声明下面的权限：

NFC

允许应用程序通过NFC发送数据。为声明该权限，要添加下面的标签作为一个 `<manifest>` 标签的子标签：

```
<uses-permission android:name="android.permission.NFC" />
```

READ_EXTERNAL_STORAGE

允许应用读取外部存储。为声明该权限，要添加下面的标签作为一个 `<manifest>` 标签的子标签：

```
<uses-permission
          android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

> **Note**：对于Android 4.2.2（API Level 17）及之前版本的系统，这个权限不是必需的。在后续版本的系统中，若应用程序需要读取外部存储，可能会需要申明该权限。为保证将来程序稳定性，建议在该权限申明变成必需的之前，先在清单文件中声明。

指定NFC功能

通过添加 `<uses-feature>` 标签作为一个 `<manifest>` 标签的子标签，指定我们的应用程序使用NFC。设置 `android:required` 属性字段为 `true`，使得我们的应用程序只有在NFC可以使用时才能运行。

下面的代码展示了如何指定 `<uses-feature>` 标签：

```
<uses-feature
    android:name="android.hardware.nfc"
    android:required="true" />
```

注意，如果应用程序将NFC作为一个可选的功能，期望在NFC不可使用时程序还能继续执行，我们就应该将 `android:required` 属性字段设为 `false`，然后在代码中测试NFC的可用性。

指定Android Beam文件传输

由于Android Beam文件传输只能在Android 4.1（API Level 16）及以上的平台使用，如果应用将Android Beam文件传输作为一个不可缺少的核心模块，那么我们必须指定 `<uses-sdk>` 标签为：android:minSdkVersion="16"。或者可以将android:minSdkVersion设置为其它值，然后在代码中测试平台版本，这部分内容将在下一节中展开。

测试设备是否支持Android Beam文件传输

应使用以下标签使得在Manifest清单文件中指定NFC是可选的：

```
<uses-feature android:name="android.hardware.nfc" android:required="false" />
```

如果设置了android:required="false"，则我们必须在代码中测试设备是否支持NFC和Android Beam文件传输。

为在代码中测试是否支持Android Beam文件传输，我们先通过PackageManager.hasSystemFeature()和参数FEATURE_NFC测试设备是否支持NFC。下一步，通过SDK_INT的值测试系统版本是否支持Android Beam文件传输。如果设备支持Android Beam文件传输，那么获得一个NFC控制器的实例，它能允许我们与NFC硬件进行通信，如下所示：

```
public class MainActivity extends Activity {
    ...
    NfcAdapter mNfcAdapter;
    // Flag to indicate that Android Beam is available
    boolean mAndroidBeamAvailable  = false;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // NFC isn't available on the device
        if (!PackageManager.hasSystemFeature(PackageManager.FEATURE_NFC)) {
            /*
             * Disable NFC features here.
             * For example, disable menu items or buttons that activate
             * NFC-related features
             */
            ...
        // Android Beam file transfer isn't supported
        } else if (Build.VERSION.SDK_INT <
                Build.VERSION_CODES.JELLY_BEAN_MR1) {
            // If Android Beam isn't available, don't continue.
            mAndroidBeamAvailable = false;
            /*
             * Disable Android Beam file transfer features here.
             */
            ...
        // Android Beam file transfer is available, continue
        } else {
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        ...
        }
    }
    ...
}
```

创建一个提供文件的回调函数

一旦确认了设备支持Android Beam文件传输，那么可以添加一个回调函数，当Android Beam文件传输监测到用户希望向另一个支持NFC的设备发送文件时，系统就会调用该函数。在该回调函数中，返回一个Uri对象数组，Android Beam文件传输会将URI对应的文件拷贝给要接收这些文件的设备。

要添加这个回调函数，需要实现NfcAdapter.CreateBeamUrisCallback接口，和它的方法：createBeamUris()，下面是一个例子：

```
public class MainActivity extends Activity {
    ...
    // List of URIs to provide to Android Beam
    private Uri[] mFileUris = new Uri[10];
    ...
    /**
     * Callback that Android Beam file transfer calls to get
     * files to share
     */
    private class FileUriCallback implements
            NfcAdapter.CreateBeamUrisCallback {
        public FileUriCallback() {
        }
        /**
         * Create content URIs as needed to share with another device
         */
        @Override
        public Uri[] createBeamUris(NfcEvent event) {
            return mFileUris;
        }
    }
    ...
}
```

一旦实现了这个接口，通过调用setBeamPushUrisCallback()将回调函数提供给Android Beam文件传输。下面是一个例子：

```
public class MainActivity extends Activity {
    ...
    // Instance that returns available files from this app
    private FileUriCallback mFileUriCallback;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Android Beam file transfer is available, continue
        ...
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        /*
         * Instantiate a new FileUriCallback to handle requests for
         * URIs
         */
        mFileUriCallback = new FileUriCallback();
        // Set the dynamic callback for URI requests.
        mNfcAdapter.setBeamPushUrisCallback(mFileUriCallback,this);
        ...
    }
    ...
}
```

> **Note**：我们也可以将Uri对象数组通过应用程序的NfcAdapter实例，直接提供给NFC框架。如果能在NFC触碰事件发生之前，定义这些URI，那么可以选择使用这个方法。更多关于这个方法的知识，请参考：NfcAdapter.setBeamPushUris()。

指定要发送的文件

为了将一或多个文件发送给其他支持NFC的设备，需要为每一个文件获取一个File URI（一个具有文件格式（file scheme）的URI），然后将它们添加至一个Uri对象数组中。此外，要传输一个文件，我们必须也拥有该文件的读权限。下例展示了如何根据文件名获取其File URI，然后将URI添加至数组当中：

```
/*
 * Create a list of URIs, get a File,
 * and set its permissions
 */
private Uri[] mFileUris = new Uri[10];
String transferFile = "transferimage.jpg";
File extDir = getExternalFilesDir(null);
File requestFile = new File(extDir, transferFile);
requestFile.setReadable(true, false);
// Get a URI for the File and add it to the list of URIs
fileUri = Uri.fromFile(requestFile);
if (fileUri != null) {
    mFileUris[0] = fileUri;
} else {
    Log.e("My Activity", "No File URI available for file.");
}
```

接收其他设备的文件

> 编写:jdneo - 原文:http://developer.android.com/training/beam-files/receive-files.html

Android Beam文件传输将文件拷贝至接收设备上的某个特殊目录。同时使用Android Media Scanner扫描拷贝的文件，并在MediaStore provider中为媒体文件添加对应的条目记录。本课将展示当文件拷贝完成时要如何响应，以及在接收设备上应该如何定位拷贝的文件。

响应请求并显示数据

当Android Beam文件传输将文件拷贝至接收设备后，它会发布一个包含Intent的通知，该Intent拥有：ACTION_VIEW，首个被传输文件的MIME类型，以及一个指向第一个文件的URI。用户点击该通知后，Intent会被发送至系统。为了使我们的应用程序能够响应该Intent，我们需要为响应的Activity所对应的<activity>标签添加一个 `<intent-filter>` 标签，在 `<intent-filter>` 标签中，添加以下子标签：

`<action android:name="android.intent.action.VIEW" />`

该标签用来匹配从通知发出的Intent，这些Intent具有ACTION_VIEW这一Action。

`<category android:name="android.intent.category.CATEGORY_DEFAULT" />`

该标签用来匹配不含有显式Category的Intent对象。

`<data android:mimeType="mime-type" />`

该标签用来匹配一个MIME类型。仅仅指定那些我们的应用能够处理的类型。

下例展示了如何添加一个intent filter来激活我们的activity：

```
    <activity
        android:name="com.example.android.nfctransfer.ViewActivity"
        android:label="Android Beam Viewer" >
        ...
        <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT"/>
            ...
        </intent-filter>
    </activity>
```

> **Note**：Android Beam文件传输不是含有ACTION_VIEW的Intent的唯一可能发送者。在接收设备上的其它应用也有可能会发送含有该Action的intent。我们马上会进一步讨论这一问题。

请求文件读权限

要读取Android Beam文件传输所拷贝到设备上的文件，需要请求READ_EXTERNAL_STORAGE权限。例如：

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

如果希望将文件拷贝至应用程序自己的存储区，那么需要的权限改为WRITE_EXTERNAL_STORAGE，另外，WRITE_EXTERNAL_STORAGE权限包含了READ_EXTERNAL_STORAGE权限。

> **Note**：对于Android 4.2.2（API Level 17）及之前版本的系统，READ_EXTERNAL_STORAGE权限仅在用户选择要读文件时才是强制需要的。而在今后的版本中会在所有情况下都需要该权限。为保证应用程序在未来的稳定性，建议在Manifest清单文件中声明该权限。

由于我们的应用对于自身的内部存储区域具有控制权，因此当要将文件拷贝至应用程序自身的的内部存储区域时，不需要声明写权限。

获取拷贝文件的目录

Android Beam文件传输一次性将所有文件拷贝到目标设备的一个目录中，Android Beam文件传输通知所发出的Intent中含有指向了第一个被传输的文件的URI。然而，我们的应用程序也有可能接收到除了Android Beam文件传输之外的某个来源所发出的含有ACTION_VIEW这一Action的Intent。为了明确应该如何处理接收的Intent，我们要检查它的Scheme和Authority。

可以调用Uri.getScheme()获得URI的Scheme，下例展示了如何确定Scheme并对URI进行相应的处理：

```
public class MainActivity extends Activity {
    ...
    // A File object containing the path to the transferred files
    private File mParentPath;
    // Incoming Intent
    private Intent mIntent;
    ...
    /*
     * Called from onNewIntent() for a SINGLE_TOP Activity
     * or onCreate() for a new Activity. For onNewIntent(),
     * remember to call setIntent() to store the most
     * current Intent
     *
     */
    private void handleViewIntent() {
        ...
        // Get the Intent action
        mIntent = getIntent();
        String action = mIntent.getAction();
        /*
         * For ACTION_VIEW, the Activity is being asked to display data.
         * Get the URI.
         */
        if (TextUtils.equals(action, Intent.ACTION_VIEW)) {
            // Get the URI from the Intent
            Uri beamUri = mIntent.getData();
            /*
             * Test for the type of URI, by getting its scheme value
             */
            if (TextUtils.equals(beamUri.getScheme(), "file")) {
                mParentPath = handleFileUri(beamUri);
            } else if (TextUtils.equals(
                    beamUri.getScheme(), "content")) {
                mParentPath = handleContentUri(beamUri);
            }
        }
        ...
    }
    ...
}
```

从File URI中获取目录

如果接收的Intent包含一个File URI，则该URI包含了一个文件的绝对文件名，它包括了完整的路径和文件名。对Android Beam文件传输来说，目录路径指向了其它被传输文件的位置（如果有其它传输文件的话），要获得该目录路径，需要取得URI的路径部分（URI中除去“file:”前缀的部分），根据路径创建一个File对象，然后获取这个File的父目录：

```
...
public String handleFileUri(Uri beamUri) {
    // Get the path part of the URI
    String fileName = beamUri.getPath();
    // Create a File object for this filename
    File copiedFile = new File(fileName);
    // Get a string containing the file's parent directory
    return copiedFile.getParent();
}
...
```

从Content URI获取目录

如果接收的Intent包含一个Content URI，这个URI可能指向的是存储于MediaStore Content Provider的目录和文件名。我们可以通过检测URI的Authority值来判断它是否是来自于MediaStore的Content URI。一个MediaStore的Content URI可能来自Android Beam文件传输也可能来自其它应用程序，但不管怎么样，我们都能根据该Content URI获得一个目录路径和文件名。

我们也可以接收一个含有ACTION_VIEW这一Action的Intent，它包含的Content URI针对于Content Provider，而不是MediaStore，这种情况下，该Content URI不包含MediaStore的Authority，且这个URI一般不指向一个目录。

> **Note**：对于Android Beam文件传输，接收在含有ACTION_VIEW的Intent中的Content URI时，若第一个接收的文件MIME类型为“audio/”，*“image/”*或者“video/*”，Android Beam文件传输会在它存储传输文件的目录内运行Media Scanner，以此为媒体文件添加索引。同时Media Scanner将结果写入MediaStore的Content Provider，之后它将第一个文件的Content URI回递给Android Beam文件传输。这个Content URI就是我们在通知Intent中所接收到的。要获得第一个文件的目录，需要使用该Content URI从MediaStore中获取它。

确定Content Provider

为了确定是否能从Content URI中获取文件目录，可以通过调用Uri.getAuthority()获取URI的Authority，以此确定与该URI相关联的Content Provider。其结果有两个可能的值：

MediaStore.AUTHORITY

表明该URI关联了被MediaStore记录的一个文件或者多个文件。可以从MediaStore中获取文件的全名，目录名就自然可以从文件全名中获取。

其他值

来自其他Content Provider的Content URI。可以显示与该Content URI相关联的数据，但是不要尝试去获取文件目录。

要从MediaStore的Content URI中获取目录，我们需要执行一个查询操作，它将Uri参数指定为收到的ContentURI，将MediaColumns.DATA列作为投影（Projection）。返回的Cursor对象包含了URI所代表的文件的完整路径和文件名。该目录路径下还包含了由Android Beam文件传输传送到该设备上的其它文件。

下面的代码展示了如何测试Content URI的Authority，并获取传输文件的路径和文件名：

```
...
public String handleContentUri(Uri beamUri) {
    // Position of the filename in the query Cursor
    int filenameIndex;
    // File object for the filename
    File copiedFile;
    // The filename stored in MediaStore
    String fileName;
    // Test the authority of the URI
    if (!TextUtils.equals(beamUri.getAuthority(), MediaStore.AUTHORITY)) {
        /*
         * Handle content URIs for other content providers
         */
    // For a MediaStore content URI
    } else {
        // Get the column that contains the file name
        String[] projection = { MediaStore.MediaColumns.DATA };
        Cursor pathCursor =
                getContentResolver().query(beamUri, projection,
                null, null, null);
        // Check for a valid cursor
        if (pathCursor != null &&
                pathCursor.moveToFirst()) {
            // Get the column index in the Cursor
            filenameIndex = pathCursor.getColumnIndex(
                    MediaStore.MediaColumns.DATA);
            // Get the full file name including path
            fileName = pathCursor.getString(filenameIndex);
            // Create a File object for the filename
            copiedFile = new File(fileName);
            // Return the parent directory of the file
            return new File(copiedFile.getParent());
         } else {
            // The query didn't work; return null
            return null;
         }
    }
}
...
```

更多关于从Content Provider获取数据的知识，请参考：Retrieving Data from the Provider。

Android多媒体

> 编写:kesenhoo - 原文:http://developer.android.com/training/building-multimedia.html

下面的这些课程会教你如何创建更加符合用户期待的富媒体的应用。

管理音频播放(Managing Audio Playback)

如何响应音频硬件按钮的点击事件，在播放音频的时候请求audio focus，以及如何正确的响应audio focus的改变。

拍照(Capturing Photos)

如何利用以及存在的相机应用进行拍照，如何直接控制相机硬件实现你自己的相机应用。

打印(Printing Content)

如何打印照片，HTML文档，自定义的文档。

管理音频播放

> 编写:kesenhoo - 原文:http://developer.android.com/training/managing-audio/index.html

如果我们的应用能够播放音频，那么让用户能够以自己预期的方式控制音频是很重要的。为了保证良好的用户体验，我们应该让应用能够管理当前的音频焦点，因为这样才能确保多个应用不会在同一时刻一起播放音频。

在学习本系列课程中，我们将会创建可以对音量按钮进行响应的应用，该应用会在播放音频的时候请求获取音频焦点，并且在当前音频焦点被系统或其他应用所改变的时候，做出正确的响应。

Lessons

- 控制音量与音频播放**(Controlling Your App's Volume and Playback)**

  学习如何确保用户能通过硬件或软件音量控制器调节应用的音量（通常这些控制器上还具有播放、停止、暂停、跳过以及回放等功能按键）。

- 管理音频焦点**(Managing Audio Focus)**

  由于可能会有多个应用具有播放音频的功能，考虑他们如何交互非常重要。为了防止多个音乐应用同时播放音频，Android使用音频焦点（Audio Focus）来控制音频的播放。在这节课中可以学习如何请求音频焦点，监听音频焦点的丢失，以及在这种情况发生时应该如何做出响应。

- 兼容音频输出设备**(Dealing with Audio Output Hardware)**

  音频有多种输出设备，在这节课中可以学习如何找出播放音频的设备，以及处理播放时耳机被拔出的情况。

控制音量与音频播放

> 编写:kesenhoo - 原文:http://developer.android.com/training/managing-audio/volume-playback.html

良好的用户体验应该是可预期且可控的。如果我们的应用可以播放音频，那么显然我们需要做到能够通过硬件按钮，软件按钮，蓝牙耳麦等来控制音量。 同样地，我们需要能够对应用的音频流进行播放（Play），停止（Stop），暂停（Pause），跳过（Skip），以及回放（Previous）等动作，并且并确保其正确性。

鉴别使用的是哪个音频流(Identify Which Audio Stream to Use)

为了创建一个良好的音频体验，我们首先需要知道应用会使用到哪些音频流。Android为播放音乐，闹铃，通知铃，来电声音，系统声音，打电话声音与拨号声音分别维护了一个独立的音频流。这样做的主要目的是让用户能够单独地控制不同的种类的音频。上述音频种类中，大多数都是被系统限制。例如，除非你的应用需要做替换闹钟的铃声的操作，不然的话你只能通过STREAM_MUSIC来播放你的音频。

使用硬件音量键来控制应用的音量(Use Hardware Volume Keys to Control Your App's Audio Volume)

默认情况下，按下音量控制键会调节当前被激活的音频流，如果我们的应用当前没有播放任何声音，那么按下音量键会调节响铃的音量。对于游戏或者音乐播放器而言，即使是在歌曲之间无声音的状态，或是当前游戏处于无声的状态，用户按下音量键的操作通常都意味着他们希望调节游戏或者音乐的音量。你可能希望通过监听音量键被按下的事件，来调节音频流的音量。其实我们不必这样做。Android提供了setVolumeControlStream()方法来直接控制指定的音频流。在鉴别出应用会使用哪个音频流之后，我们需要在应用生命周期的早期阶段调用该方法，因为该方法只需要在Activity整个生命周期中调用一次，通常，我们可以在负责控制多媒体的Activity或者Fragment的 `onCreate()` 方法中调用它。这样能确保不管应用当前是否可见，音频控制的功能都能符合用户的预期。

```
setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

自此之后，不管目标Activity或Fragment是否可见，按下设备的音量键都能够影响我们指定的音频流（在这个例子中，音频流是"music"）。

使用硬件的播放控制按键来控制应用的音频播放(Use Hardware Playback Control Keys to Control Your App's Audio Playback)

许多线控或者无线耳机都会有许多媒体播放控制按钮，例如：播放，停止，暂停，跳过，以及回放等。无论用户按下设备上任意一个控制按钮，系统都会广播一个带有ACTION_MEDIA_BUTTON的Intent。为了正确地响应这些操作，需要在Manifest文件中注册一个针对于该Action的BroadcastReceiver，如下所示：

```
<receiver android:name=".RemoteControlReceiver">
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_BUTTON" />
    </intent-filter>
</receiver>
```

在Receiver的实现中，需要判断这个广播来自于哪一个按钮，Intent通过EXTRA_KEY_EVENT这一Key包含了该信息，另外，KeyEvent类包含了一系列诸如 `KEYCODE_MEDIA_*` 的静态变量来表示不同的媒体按钮，例如KEYCODE_MEDIA_PLAY_PAUSE 与 KEYCODE_MEDIA_NEXT。

```
public class RemoteControlReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_MEDIA_BUTTON.equals(intent.getAction())) {
            KeyEvent event = (KeyEvent)intent.getParcelableExtra(Intent.EXTRA_KEY_EVEN
T);
            if (KeyEvent.KEYCODE_MEDIA_PLAY == event.getKeyCode()) {
                // Handle key press.
            }
        }
    }
}
```

因为可能会有多个程序在监听与媒体按钮相关的事件，所以我们必须在代码中控制应用接收相关事件的时机。下面的例子显示了如何使用AudioManager来为我们的应用注册监听与取消监听媒体按钮事件，当Receiver被注册上时，它将是唯一一个能够响应媒体按钮广播的Receiver。

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Start listening for button presses
am.registerMediaButtonEventReceiver(RemoteControlReceiver);
...

// Stop listening for button presses
am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
```

通常，应用需要在他们失去焦点或者不可见的时候（比如在onStop()方法里面）取消注册监听。但是对于媒体播放应用来说并没有那么简单，实际上，在应用不可见（不能通过可见的UI控件进行控制）的时候，仍然能够响应媒体播放按钮事件是极其重要的。为了实现这一点，有一个更好的方法，我们可以在程序获取与失去音频焦点的时候注册与取消对音频按钮事件的监听。这个内容会在后面的课程中详细讲解。

管理音频焦点

> 编写:kesenhoo - 原文:http://developer.android.com/training/managing-audio/audio-focus.html

由于可能会有多个应用可以播放音频，所以我们应当考虑一下他们应该如何交互。为了防止多个音乐播放应用同时播放音频，Android使用音频焦点（Audio Focus）来控制音频的播放——即只有获取到音频焦点的应用才能够播放音频。

在我们的应用开始播放音频之前，它需要先请求音频焦点，然后再获取到音频焦点。另外，它还需要知道如何监听失去音频焦点的事件并对此做出合适的响应。

请求获取音频焦点(Request the Audio Focus)

在我们的应用开始播放音频之前，它需要获取将要使用的音频流的音频焦点。通过使用requestAudioFocus() 方法可以获取我们希望得到的音频流焦点。如果请求成功，该方法会返回AUDIOFOCUS_REQUEST_GRANTED。

另外我们必须指定正在使用的音频流，而且需要确定所请求的音频焦点是短暂的（Transient）还是永久的（Permanent）。

- 短暂的焦点锁定：当计划播放一个短暂的音频时使用（比如播放导航指示）。
- 永久的焦点锁定：当计划播放一个较长但时长可预期的音频时使用（比如播放音乐）。

下面的代码片段是一个在播放音乐时请求永久音频焦点的例子，我们必须在开始播放之前立即请求音频焦点，比如在用户点击播放或者游戏中下一关的背景音乐开始前。

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                                 // Use the music stream.
                                 AudioManager.STREAM_MUSIC,
                                 // Request permanent focus.
                                 AudioManager.AUDIOFOCUS_GAIN);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    am.registerMediaButtonEventReceiver(RemoteControlReceiver);
    // Start playback.
}
```

一旦结束了播放，需要确保调用了abandonAudioFocus()方法。这样相当于告知系统我们不再需要获取焦点并且注销所关联的AudioManager.OnAudioFocusChangeListener监听器。对于另一种释放短暂音频焦点的情况，这会允许任何被我们打断的应用可以继续播放。

```
// Abandon audio focus when playback complete
am.abandonAudioFocus(afChangeListener);
```

当请求短暂音频焦点的时候，我们可以选择是否开启“Ducking”。通常情况下，一个应用在失去音频焦点时会立即关闭它的播放声音。如果我们选择在请求短暂音频焦点的时候开启了Ducking，那意味着其它应用可以继续播放，仅仅是在这一刻降低自己的音量，直到重新获取到音频焦点后恢复正常音量（译注：也就是说，不用理会这个短暂焦点的请求，这并不会打断目前正在播放的音频。比如在播放音乐的时候突然出现一个短暂的短信提示声音，此时仅仅是把歌曲的音量暂时调低，使得用户能够听到短信提示声，在此之后便立马恢复正常播放）。

```
// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                             // Use the music stream.
                             AudioManager.STREAM_MUSIC,
                             // Request permanent focus.
                             AudioManager.AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    // Start playback.
}
```

Ducking对于那些间歇性使用音频焦点的应用来说特别合适，比如语音导航。

如果有另一个应用像上述那样请求音频焦点，它所请求的永久音频焦点或者短暂音频焦点（支持Ducking或不支持Ducking），都会被你在请求获取音频焦点时所注册的监听器接收到。

处理失去音频焦点(Handle the Loss of Audio Focus)

如果应用A请求获取了音频焦点，那么在应用B请求获取音频焦点的时候，A获取到的焦点就会失去。如何响应失去焦点事件，取决于失去焦点的方式。

在音频焦点的监听器里面，当接受到描述焦点改变的事件时会触发onAudioFocusChange()回调方法。如之前提到的，获取焦点有三种类型，我们同样会有三种失去焦点的类型：永久失去，短暂失去，允许Ducking的短暂失去。

- 失去短暂焦点：通常在失去短暂焦点的情况下，我们会暂停当前音频的播放或者降低音量，同时需要准备在重新获取到焦点之后恢复播放。

- 失去永久焦点：假设另外一个应用开始播放音乐，那么我们的应用就应该有效地将自己停止。在实际场景当中，这意味着停止播放，移除媒体按钮监听，允许新的音频播放器可以唯一地监听那些按钮事件，并且放弃自己的音频焦点。此时，如果想要恢复自己的音频播放，我们需要等待某种特定用户行为发生（例如按下了我们应用当中的播放按钮）。

在下面的代码片段当中，如果焦点的失去是短暂型的，我们将音频播放对象暂停，并在重新获取到焦点后进行恢复。如果是永久型的焦点失去事件，那么我们的媒体按钮监听器会被注销，并且不再监听音频焦点的改变。

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT
            // Pause playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Resume playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_LOSS) {
            am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
            am.abandonAudioFocus(afChangeListener);
            // Stop playback
        }
    }
};
```

在上面失去短暂焦点的例子中，如果允许Ducking，那么除了暂停当前的播放之外，我们还可以选择使用“Ducking”。

Duck!

在使用Ducking时，正常播放的歌曲会降低音量来凸显这个短暂的音频声音，这样既让这个短暂的声音比较突出，又不至于打断正常的声音。

下面的代码片段让我们的播放器在暂时失去音频焦点时降低音量，并在重新获得音频焦点之后恢复原来音量。

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK) {
            // Lower the volume
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Raise it back to normal
        }
    }
};
```

音频焦点的失去是我们需要响应的最重要的事件广播之一，但除此之外还有很多其他重要的广播需要我们正确地做出响应。系统会广播一系列的Intent来向你告知用户在使用音频过程当中的各种变化。下节课会演示如何监听这些广播并提升用户的整体体验。

兼容音频输出设备

> 编写:kesenhoo - 原文:http://developer.android.com/training/managing-audio/audio-output.html

当用户想要通过Android设备欣赏音乐的时候，他可以有多种选择，大多数设备拥有内置的扬声器，有线耳机，也有其它很多设备支持蓝牙连接，有些甚至还支持A2DP蓝牙音频传输模型协定。（译注：A2DP全名是Advanced Audio Distribution Profile 蓝牙音频传输模型协定! A2DP是能够采用耳机内的芯片来堆栈数据，达到声音的高清晰度。有A2DP的耳机就是蓝牙立体声耳机。声音能达到44.1kHz，一般的耳机只能达到8kHz。如果手机支持蓝牙，只要装载A2DP协议，就能使用A2DP耳机了。还有消费者看到技术参数提到蓝牙V1.0 V1.1 V1.2 V2.0 - 这些是指蓝牙的技术版本，是指通过蓝牙传输的速度，他们是否支持A2DP具体要看蓝牙产品制造商是否使用这个技术。来自百度百科）

检测目前正在使用的硬件设备(Check What Hardware is Being Used)

使用不同的硬件播放声音会影响到应用的行为。可以使用AudioManager来查询当前音频是输出到扬声器，有线耳机还是蓝牙上，如下所示：

```
if (isBluetoothA2dpOn()) {
    // Adjust output for Bluetooth.
} else if (isSpeakerphoneOn()) {
    // Adjust output for Speakerphone.
} else if (isWiredHeadsetOn()) {
    // Adjust output for headsets
} else {
    // If audio plays and noone can hear it, is it still playing?
}
```

处理音频输出设备的改变(Handle Changes in the Audio Output Hardware)

当有线耳机被拔出或者蓝牙设备断开连接的时候，音频流会自动输出到内置的扬声器上。假设播放声音很大，这个时候突然转到扬声器播放会显得非常嘈杂。

幸运的是，系统会在这种情况下广播带有ACTION_AUDIO_BECOMING_NOISY的Intent。无论何时播放音频，我们都应该注册一个BroadcastReceiver来监听这个Intent。在使用音乐播放器时，用户通常会希望此时能够暂停当前歌曲的播放。而在游戏当中，用户通常会希望可以减低音量。

```
private class NoisyAudioStreamReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (AudioManager.ACTION_AUDIO_BECOMING_NOISY.equals(intent.getAction())) {
            // Pause the playback
        }
    }
}

private IntentFilter intentFilter = new IntentFilter(AudioManager.ACTION_AUDIO_BECOMIN
G_NOISY);

private void startPlayback() {
    registerReceiver(myNoisyAudioStreamReceiver(), intentFilter);
}

private void stopPlayback() {
    unregisterReceiver(myNoisyAudioStreamReceiver);
}
```

拍照

> 编写:kesenhoo - 原文:http://developer.android.com/training/camera/index.html

在多媒体技术还未流行之时，我们的世界并不像现在这样多姿多彩。还记得Gopher吗？（*Gopher*是计算机上的一个工具软件，是*Internet*提供的一种由菜单式驱动的信息查询工具，采用客户机/服务器模式）。如果我们希望将我们的应用变成用户生活的一部分，那么我们应该给用户提供一种方式，让他们可以将自己的生活融入到我们的应用中来。通过相机，我们的应用可以让用户扩展他们所看到的事物：生成唯一的头像，通过相机玩寻找僵尸的交互性游戏，亦或者是分享他们的某些经历。

这一章节，我们会学习如何简单地借助于已经存在的相机应用，完成一些特定的功能。在后面的课程中，我们还会更加深入地学习如何直接控制相机硬件。

样例代码

PhotoIntentActivity.zip

Lessons

- 轻松拍摄照片

  用仅仅几行代码调用其他应用拍照。

- 轻松录制视频

  用仅仅几行代码调用其他应用录像。

- 控制相机

  直接控制相机硬件，实现你自己的相机应用。

轻松拍摄照片

> 编写:kesenhoo - 原文:http://developer.android.com/training/camera/photobasics.html

这节课将讲解如何使用已有的相机应用拍摄照片。

假设我们正在实现一个基于人群的气象服务，通过应用客户端拍下的天气图片汇聚在一起，可以组成全球气象图。整合图片只是应用的一小部分，我们想要通过最简单的方式获取图片，而不是重新设计并实现一个具有相机功能的组件。幸运的是，通常来说，大多数Android设备都已经安装了至少一款相机程序。在这节课中，我们会学习如何利用已有的相机应用拍摄照片。

请求使用相机权限

如果拍照是应用的必要功能，那么应该令它在Google Play中仅对有相机的设备可见。为了让用户知道我们的应用需要依赖相机，在Manifest清单文件中添加 `<uses-feature>` 标签:

```
 <manifest ... >
    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
    ...
</manifest>
```

如果我们的应用使用相机，但相机并不是应用的正常运行所必不可少的组件，可以将 `android:required` 设置为 `"false"` 。这样的话，Google Play 也会允许没有相机的设备下载该应用。当然我们有必要在使用相机之前通过调用hasSystemFeature(PackageManager.FEATURE_CAMERA)方法来检查设备上是否有相机。如果没有，我们应该禁用和相机相关的功能！

使用相机应用程序进行拍照

利用一个描述了执行目的Intent对象，Android可以将某些执行任务委托给其他应用。整个过程包含三部分： Intent 本身，一个函数调用来启动外部的 Activity，当焦点返回到我们的Activity时，处理返回图像数据的代码。

下面的函数通过发送一个Intent来捕获照片：

```
static final int REQUEST_IMAGE_CAPTURE = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE);
    }
}
```

注意在调用startActivityForResult()方法之前，先调用resolveActivity()，这个方法会返回能处理该Intent的第一个Activity（译注：即检查有没有能处理这个Intent的Activity）。执行这个检查非常重要，因为如果在调用startActivityForResult()时，没有应用能处理你的Intent，应用将会崩溃。所以只要返回结果不为null，使用该Intent就是安全的。

获取缩略图

拍摄照片并不是应用的最终目的，我们还想要从相机应用那里取回拍摄的照片，并对它执行某些操作。

Android的相机应用会把拍好的照片编码为缩小的Bitmap，使用extra value的方式添加到返回的Intent当中，并传送给onActivityResult()，对应的Key为 `"data"` 。下面的代码展示的是如何获取这一图片并显示在ImageView上。

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
        mImageView.setImageBitmap(imageBitmap);
    }
}
```

> **Note:** 这张从 `"data"` 中取出的缩略图适用于作为图标，但其他作用会比较有限。而处理一张全尺寸图片需要做更多的工作。

保存全尺寸照片

如果我们提供了一个File对象给Android的相机程序，它会保存这张全尺寸照片到给定的路径下。另外，我们必须提供存储图片所需要的含有后缀名形式的文件名。

一般而言，用户使用设备相机所拍摄的任何照片都应该被存放在设备的公共外部存储中，这样它们就能被所有的应用访问。将DIRECTORY_PICTURES作为参数，传递给getExternalStoragePublicDirectory()方法，可以返回适用于存储公共图片的目录。由于该方法提供的目录被所有应用共享，因此对该目录进行读写操作分别需要READ_EXTERNAL_STORAGE和WRITE_EXTERNAL_STORAGE权限。另外，因为写权限隐含了读权限，所以如果需要外部存储的写权限，那么仅仅需要请求一项权限就可以了：

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

然而，如果希望照片对我们的应用而言是私有的，那么可以使用getExternalFilesDir()提供的目录。在Android 4.3及以下版本的系统中，写这个目录需要WRITE_EXTERNAL_STORAGE权限。从Android 4.4开始，该目录将无法被其他应用访问，所以该权限就不再需要了，你可以通过添加maxSdkVersion属性，声明只在低版本的Android设备上请求这个权限。

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

> **Note:** 所有存储在getExternalFilesDir()提供的目录中的文件会在用户卸载你的app后被删除。

一旦选定了存储文件的目录，我们还需要设计一个保证文件名不会冲突的命名规则。当然我们还可以将路径存储在一个成员变量里以备在将来使用。下面的例子使用日期时间戳作为新照片的文件名：

```
String mCurrentPhotoPath;

private File createImageFile() throws IOException {
    // Create an image file name
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageFileName = "JPEG_" + timeStamp + "_";
    File storageDir = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES);
    File image = File.createTempFile(
        imageFileName,  /* prefix */
        ".jpg",         /* suffix */
        storageDir      /* directory */
    );

    // Save a file: path for use with ACTION_VIEW intents
    mCurrentPhotoPath = "file:" + image.getAbsolutePath();
    return image;
}
```

有了上面的方法，我们就可以给新照片创建文件对象了，现在我们可以像这样创建并触发一个Intent：

```
static final int REQUEST_TAKE_PHOTO = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    // Ensure that there's a camera activity to handle the intent
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        // Create the File where the photo should go
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            // Error occurred while creating the File
            ...
        }
        // Continue only if the File was successfully created
        if (photoFile != null) {
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
                    Uri.fromFile(photoFile));
            startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
        }
    }
}
```

将照片添加到相册中

由于我们通过Intent创建了一张照片，因此图片的存储位置我们是知道的。对其他人来说，也许查看我们的照片最简单的方式是通过系统的Media Provider。

> **Note:** 如果将图片存储在getExternalFilesDir()提供的目录中，Media Scanner将无法访问到我们的文件，因为它们隶属于应用的私有数据。

下面的例子演示了如何触发系统的Media Scanner，将我们的照片添加到Media Provider的数据库中，这样就可以使得Android相册程序与其他程序能够读取到这些照片。

```
private void galleryAddPic() {
    Intent mediaScanIntent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    File f = new File(mCurrentPhotoPath);
    Uri contentUri = Uri.fromFile(f);
    mediaScanIntent.setData(contentUri);
    this.sendBroadcast(mediaScanIntent);
}
```

解码一幅缩放图片

在有限的内存下，管理许多全尺寸的图片会很棘手。如果发现应用在展示了少量图片后消耗了所有内存，我们可以通过缩放图片到目标视图尺寸，之后再载入到内存中的方法，来显著降低内存的使用，下面的例子演示了这个技术：

```
private void setPic() {
    // Get the dimensions of the View
    int targetW = mImageView.getWidth();
    int targetH = mImageView.getHeight();

    // Get the dimensions of the bitmap
    BitmapFactory.Options bmOptions = new BitmapFactory.Options();
    bmOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    int photoW = bmOptions.outWidth;
    int photoH = bmOptions.outHeight;

    // Determine how much to scale down the image
    int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

    // Decode the image file into a Bitmap sized to fill the View
    bmOptions.inJustDecodeBounds = false;
    bmOptions.inSampleSize = scaleFactor;
    bmOptions.inPurgeable = true;

    Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    mImageView.setImageBitmap(bitmap);
}
```

轻松录制视频

> 编写:kesenhoo - 原文:http://developer.android.com/training/camera/videobasics.html

这节课会介绍如何使用已有的相机应用来录制视频。

假设在我们应用的所有功能当中，整合视频只是其中的一小部分，我们想要以最简单的方法录制视频，而不是重新实现一个摄像机组件。幸运的是，大多数Android设备已经安装了一个能录制视频的相机应用。在本节课当中，我们将会让它为我们完成这一任务。

请求相机权限

为了让用户知道我们的应用依赖照相机，在Manifest清单文件中添加 `<uses-feature>` 标签:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
    ...
</manifest>
```

如果应用使用相机，但相机并不是应用正常运行所必不可少的组件，可以将 `android:required` 设置为 `"false"` 。这样的话，Google Play 也会允许没有相机的设备下载该应用。当然我们有必要在使用相机之前通过调用hasSystemFeature(PackageManager.FEATURE_CAMERA)方法来检查设备上是否有相机。如果没有，那么和相机相关的功能应该禁用！

使用相机程序来录制视频

利用一个描述了执行目的的Intent对象，Android可以将某些执行任务委托给其他应用。整个过程包含三部分： Intent 本身，一个函数调用来启动外部的 Activity，当焦点返回到Activity时，处理返回图像数据的代码。

下面的函数将会发送一个Intent来录制视频

```
static final int REQUEST_VIDEO_CAPTURE = 1;

private void dispatchTakeVideoIntent() {
    Intent takeVideoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    if (takeVideoIntent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(takeVideoIntent, REQUEST_VIDEO_CAPTURE);
    }
}
```

注意在调用startActivityForResult()方法之前，先调用resolveActivity()，这个方法会返回能处理该Intent的第一个Activity（译注：即检查有没有能处理这个Intent的Activity）。执行这个检查非常重要，因为如果在调用startActivityForResult()时，没有应用能处理你的Intent，应用将会崩溃。所以只要返回结果不为null，使用该Intent就是安全的。

查看视频

Android的相机程序会把指向视频存储地址的Uri添加到Intent中，并传送给onActivityResult()方法。下面的代码获取该视频并显示到一个VideoView当中：

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_VIDEO_CAPTURE && resultCode == RESULT_OK) {
        Uri videoUri = intent.getData();
        mVideoView.setVideoURI(videoUri);
    }
}
```

控制相机

> 编写:kesenhoo@2016/11/30 - http://developer.android.com/training/camera/cameradirect.html

在这一节课，我们会讨论如何通过使用Android框架所提供的API来直接控制相机硬件，实现自定义相机模块。

直接控制相机，相比起请求已经存在的相机应用进行拍照或录制视频，要复杂一些。这节课将会讲解如何创建一个专业的相机应用并将其整合到我们自己的应用界面中去。

打开相机对象

获取一个 Camera 实例是直接控制相机的第一步。正如Android自带的Camera程序一样，推荐的方式是在Activity的onCreate()方法里面另起一个线程，在这个单独的线程里面对Camera进行操作。在单独的线程里面访问Camera实例可以避免操作Camera实例的时间较长而导致UI线程被阻塞。更基础的实现方式是，编写一个打开Camera的方法，这个方法可以在onResume()方法里面去调用执行，单独的方法使得代码更容易重用，也便于保持控制流程更加简单。

如果我们在执行Camera.open()方法的时候Camera正在被另外一个应用使用，那么函数会抛出一个Exception，我们可以利用 `try` 语句块进行捕获：

```
private boolean safeCameraOpen(int id) {
    boolean qOpened = false;

    try {
        releaseCameraAndPreview();
        mCamera = Camera.open(id);
        qOpened = (mCamera != null);
    } catch (Exception e) {
        Log.e(getString(R.string.app_name), "failed to open Camera");
        e.printStackTrace();
    }

    return qOpened;
}

private void releaseCameraAndPreview() {
    mPreview.setCamera(null);
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

自从API level 9开始，相机框架可以支持多个摄像头的打开操作。如果使用旧的API，在调用open()时不传入参数指定打开哪个摄像头，默认情况下会使用后置摄像头。

创建相机预览界面

拍照通常需要向用户提供一个预览界面来显示待拍摄的画面内容。我们可以使用SurfaceView来呈现相机采集到的图像画面。

Preview预览组件

我们需要使用preview class来显示预览界面。这个类需要实现 `android.view.SurfaceHolder.Callback` 接口，它会用这个接口把相机硬件获取到的图像数据传递给应用程序。

```
class Preview extends ViewGroup implements SurfaceHolder.Callback {

    SurfaceView mSurfaceView;
    SurfaceHolder mHolder;

    Preview(Context context) {
        super(context);

        mSurfaceView = new SurfaceView(context);
        addView(mSurfaceView);

        // Install a SurfaceHolder.Callback so we get notified when the
        // underlying surface is created and destroyed.
        mHolder = mSurfaceView.getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
...
}
```

为了能够呈现相机图像画面，Preview类必须先获取Camera实例。

设置和启动Preview

一个Camera实例与它相关的Preview必须按照特定的顺序来创建，通常来说Camera对象优先被创建。在下面的示例中，初始化Camera的动作被封装了起来，这样，无论用户想对Camera做什么样的改变，Camera.startPreview()都会被 `setCamera()` 调用。另外，Preview对象必须在 `surfaceChanged()` 这一回调方法里面重新启用（restart）。

```
public void setCamera(Camera camera) {
    if (mCamera == camera) { return; }

    stopPreviewAndFreeCamera();

    mCamera = camera;

    if (mCamera != null) {
        List<Size> localSizes = mCamera.getParameters().getSupportedPreviewSizes();
        mSupportedPreviewSizes = localSizes;
        requestLayout();

        try {
            mCamera.setPreviewDisplay(mHolder);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Important: Call startPreview() to start updating the preview
        // surface. Preview must be started before you can take a picture.
        mCamera.startPreview();
    }
}
```

修改相机设置

相机参数的修改可以改变拍照的成像效果，例如缩放大小，曝光补偿值等等。下面的例子仅仅演示了如何改变预览大小，更多设置请参考相机应用的源代码。

```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    // Now that the size is known, set up the camera parameters and begin
    // the preview.
    Camera.Parameters parameters = mCamera.getParameters();
    parameters.setPreviewSize(mPreviewSize.width, mPreviewSize.height);
    requestLayout();
    mCamera.setParameters(parameters);

    // Important: Call startPreview() to start updating the preview surface.
    // Preview must be started before you can take a picture.
    mCamera.startPreview();
}
```

设置预览方向

大多数相机程序会锁定预览方向为横屏状态，因为该方向是相机传感器的自然放置方向。当然这一设定并不妨碍我们去拍竖屏的照片，这个时候设备的方向角度信息会被记录在EXIF文件头中。setCameraDisplayOrientation()方法可以让你在不影响照片拍摄过程的情况下，改变预览的方向。然而，对于Android API level 14及更旧版本的系统，在改变方向之前，我们必须先停止相机预览，设置方向之后，然后再重启预览。

拍摄照片

一旦预览启动成功之后，可以使用Camera.takePicture()方法拍摄照片。我们可以创建Camera.PictureCallback与Camera.ShutterCallback对象并将他们传递到Camera.takePicture()中。

如果我们想要获取每一帧的相机画面，可以创建一个Camera.PreviewCallback并实现onPreviewFrame()回调。我们可以取景画面帧进行保存，也可以延迟调用takePicture()来进行拍照。

重启Preview

在拍摄好图片后，我们必须在用户拍下一张图片之前重启预览。下面的示例是根据快门按钮的不同状态来实现重启预览。

```
@Override
public void onClick(View v) {
    switch(mPreviewState) {
    case K_STATE_FROZEN:
        mCamera.startPreview();
        mPreviewState = K_STATE_PREVIEW;
        break;

    default:
        mCamera.takePicture( null, rawCallback, null);
        mPreviewState = K_STATE_BUSY;
    } // switch
    shutterBtnConfig();
}
```

停止预览并释放相机

当应用使用完相机之后，我们有必要进行清理释放资源的操作。尤其是，我们必须释放Camera对象，不然的话可能会引起其他应用程序使用Camera实例的时候发生崩溃，包括我们自己应用也同样会遇到这个问题。

那么何时应该停止预览并释放相机呢？在预览Surface组件被销毁之后，可以做停止预览与释放相机的操作。如下面Preview类中的方法所做的那样：

```
public void surfaceDestroyed(SurfaceHolder holder) {
    // Surface will be destroyed when we return, so stop the preview.
    if (mCamera != null) {
        // Call stopPreview() to stop updating the preview surface.
        mCamera.stopPreview();
    }
}

/**
 * When this function returns, mCamera will be null.
 */
private void stopPreviewAndFreeCamera() {

    if (mCamera != null) {
        // Call stopPreview() to stop updating the preview surface.
        mCamera.stopPreview();

        // Important: Call release() to release the camera for use by other
        // applications. Applications should release the camera immediately
        // during onPause() and re-open() it during onResume()).
        mCamera.release();

        mCamera = null;
    }
}
```

在这节课的前部分中，这一些系列的动作也是 `setCamera()` 方法的一部分，因此初始化一个相机的动作，总是从停止预览开始的。

打印

> 编写:jdneo - 原文:http://developer.android.com/training/printing/index.html

Android用户经常需要在设备上单独地阅览信息，但有时候也需要为了分享信息而不得不给其他人看自己设备的屏幕，这显然不是分享信息的好办法。如果我们可以通过Android应用把希望分享的信息打印出来，这将给用户提供一种从应用获取更多信息的好办法，更何况这么做还能将信息分享给其他那些不使用我们的应用的人。另外，打印服务还能创建信息的快照（生成PDF文件），而这一切不需要打印设备，无线网络连接，也不会消耗过多电量。

在Android 4.4（API Level 19）及更高版本的系统中，框架提供了直接从Android应用程序打印图片和文字的服务。这系列课程将展示如何启用打印：包括打印图片，HTML页面以及创建自定义的打印文档。

Lessons

- 打印照片

  这节课将展示如何打印一幅图像。

- 打印**HTML**文档

  这节课将展示如何打印一个HTML文档。

- 打印自定义文档

  这节课将展示如何连接到Android打印管理器，创建一个打印适配器并建立要打印的内容。

打印照片

> 编写:jdneo - 原文:http://developer.android.com/training/printing/photos.html

拍摄并分享照片是移动设备最流行的用法之一。如果我们的应用拍摄了照片，并期望可以展示他们，或者允许用户共享照片，那么我们就应该考虑让应用可以打印出这些照片来。Android Support Library提供了一个方便的函数，通过这一函数，仅仅使用很少量的代码和一些简单的打印布局配置集，就能够进行照片打印。

这堂课将展示如何使用v4 support library中的PrintHelper类打印一幅图片。

打印一幅图片

Android Support Library中的PrintHelper类提供了一种打印图片的简单方法。该类有一个单一的布局选项：setScaleMode()，它允许我们使用下面的两个选项之一：

- SCALE_MODE_FIT：该选项会调整图像的大小，这样整个图像就会在打印有效区域内全部显示出来（等比例缩放至长和宽都包含在纸张页面内）。
- SCALE_MODE_FILL：该选项同样会等比例地调整图像的大小使图像充满整个打印有效区域，即让图像充满整个纸张页面。这就意味着如果选择这个选项，那么图片的一部分（顶部和底部，或者左侧和右侧）将无法打印出来。如果不设置图像的打印布局选项，该模式将是默认的图像拉伸方式。

这两个setScaleMode()的图像布局选项都会保持图像原有的长宽比。下面的代码展示了如何创建一个PrintHelper类的实例，设置布局选项，并开始打印进程：

```
private void doPhotoPrint() {
    PrintHelper photoPrinter = new PrintHelper(getActivity());
    photoPrinter.setScaleMode(PrintHelper.SCALE_MODE_FIT);
    Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
            R.drawable.droids);
    photoPrinter.printBitmap("droids.jpg - test print", bitmap);
}
```

该方法可以作为一个菜单项的Action来被调用。注意对于那些不一定被设备支持的菜单项（比如有些设备可能无法支持打印），应该放置在“更多菜单（overflow menu）”中。要获取有关这方面的更多知识，可以阅读：Action Bar。

在printBitmap()被调用之后，我们的应用就不再需要进行其他的操作了。之后Android打印界面就会出现，允许用户选择一个打印机和它的打印选项。用户可以打印图像或者取消这一次操作。如果用户选择了打印图像，那么一个打印任务将会被创建，同时在系统的通知栏中会

显示一个打印提醒通知。

如果希望在打印输出中包含更多的内容，而不仅仅是一张图片，那么就必须构造一个打印文档。这方面知识将会在后面的两节课程中展开。

打印HTML文档

> 编写:jdneo - 原文:http://developer.android.com/training/printing/html-docs.html

如果要在Android上打印比一副照片更丰富的内容，我们需要将文本和图片组合在一个待打印的文档中。Android框架提供了一种使用HTML语言来构建文档并进行打印的方法，它使用的代码数量是很小的。

WebView类在Android 4.4（API Level 19）中得到了更新，使得它可以打印HTML内容。该类允许我们加载一个本地HTML资源或者从网页下载一个页面，创建一个打印任务，并把它交给Android打印服务。

这节课将展示如何快速地构建一个包含有文本和图片的HTML文档，以及如何使用WebView打印该文档。

加载一个HTML文档

用WebView打印一个HTML文档，会涉及到加载一个HTML资源，或者用一个字符串构建HTML文档。这一节将描述如何构建一个HTML的字符串并将它加载到WebView中，以备打印。

该View对象一般被用来作为一个Activity布局的一部分。然而，如果应用当前并没有使用WebView，我们可以创建一个该类的实例，以进行打印。创建该自定义View的主要步骤是：

1. 在HTML资源加载完毕后，创建一个WebViewClient用来启动一个打印任务。
2. 加载HTML资源至WebView对象中。

下面的代码展示了如何创建一个简单的WebViewClient并且加载一个动态创建的HTML文档：

```
private WebView mWebView;

private void doWebViewPrint() {
    // Create a WebView object specifically for printing
    WebView webView = new WebView(getActivity());
    webView.setWebViewClient(new WebViewClient() {

            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                return false;
            }

            @Override
            public void onPageFinished(WebView view, String url) {
                Log.i(TAG, "page finished loading " + url);
                createWebPrintJob(view);
                mWebView = null;
            }
    });

    // Generate an HTML document on the fly:
    String htmlDocument = "<html><body><h1>Test Content</h1><p>Testing, " +
            "testing, testing...</p></body></html>";
    webView.loadDataWithBaseURL(null, htmlDocument, "text/HTML", "UTF-8", null);

    // Keep a reference to WebView object until you pass the PrintDocumentAdapter
    // to the PrintManager
    mWebView = webView;
}
```

> **Note**： 请确保在WebViewClient)中的onPageFinished()方法内调用创建打印任务的方法。如果没有等到页面加载完毕就进行打印，打印的输出可能会不完整或空白，甚至可能会失败。
>
> **Note**：在上面的样例代码中，保留了一个WebView对象实例的引用，这样能够确保它不会在打印任务创建之前就被垃圾回收器所回收。在编写代码时请务必这样做，否则打印的进程可能会无法继续执行。

如果我们希望页面中包含图像，将这个图像文件放置在你的工程的“assets/”目录中，并指定一个基URL（Base URL），并将它作为loadDataWithBaseURL()方法的第一个参数，就像下面所显示的一样：

```
webView.loadDataWithBaseURL("file:///android_asset/images/", htmlBody,
        "text/HTML", "UTF-8", null);
```

我们也可以加载一个需要打印的网页，具体做法是将loadDataWithBaseURL()方法替换为loadUrl()，如下所示：

```
// Print an existing web page (remember to request INTERNET permission!):
webView.loadUrl("http://developer.android.com/about/index.html");
```

当使用WebView创建打印文档时，你要注意下面的一些限制：

- 不能为文档添加页眉和页脚，包括页号。
- HTML文档的打印选项不包含选择打印的页数范围，例如：对于一个10页的HTMl文档，只打印2到4页是不可以的。
- 一个WebView的实例只能在同一时间处理一个打印任务。
- 若一个HTML文档包含CSS打印属性，比如一个landscape属性，这是不被支持的。
- 不能通过一个HTML文档中的JavaScript脚本来激活打印。

> **Note**：一旦在布局中包含的WebView对象将文档加载完毕后，就可以打印WebView对象的内容了。

如果希望创建一个更加自定义化的打印输出并希望可以完全控制打印页面上绘制的内容，可以学习下一节课程：打印自定义文档

创建一个打印任务

在创建了WebView并加载了我们的HTML内容之后，应用就已经几乎完成了属于它的任务。接下来，我们需要访问PrintManager，创建一个打印适配器，并在最后创建一个打印任务。下面的代码展示了如何执行这些步骤：

```
private void createWebPrintJob(WebView webView) {

    // Get a PrintManager instance
    PrintManager printManager = (PrintManager) getActivity()
            .getSystemService(Context.PRINT_SERVICE);

    // Get a print adapter instance
    PrintDocumentAdapter printAdapter = webView.createPrintDocumentAdapter();

    // Create a print job with name and adapter instance
    String jobName = getString(R.string.app_name) + " Document";
    PrintJob printJob = printManager.print(jobName, printAdapter,
            new PrintAttributes.Builder().build());

    // Save the job object for later status checking
    mPrintJobs.add(printJob);
}
```

这个例子保存了一个PrintJob对象的实例，以供我们的应用将来使用，当然这是不必须的。我们的应用可以使用这个对象来跟踪打印任务执行时的进度。如果希望监控应用中的打印任务是否完成，是否失败或者是否被用户取消，这个方法非常有用。另外，我们不需要创建一个应用内置的通知，因为打印框架会自动的创建一个该打印任务的系统通知。

打印自定义文档

> 编写:jdneo - 原文:http://developer.android.com/training/printing/custom-docs.html

对于有些应用，比如绘图应用，页面布局应用和其它一些关注于图像输出的应用，创造出精美的打印页面将是它的核心功能。在这种情况下，仅仅打印一幅图片或一个HTML文档就不够了。这类应用的打印输出需要精确地控制每一个会在页面中显示的对象，包括字体，文本流，分页符，页眉，页脚和一些图像元素等等。

想要创建一个完全自定义的打印文档，需要投入比之前讨论的方法更多的编程精力。我们必须构建可以和打印框架相互通信的组件，调整打印参数，绘制页面元素并管理多个页面的打印。

这节课将展示如何连接打印管理器，创建一个打印适配器以及如何构建出需要打印的内容。

连接打印管理器

当我们的应用直接管理打印进程时，在收到来自用户的打印请求后，第一步要做的是连接Android打印框架并获取一个PrintManager类的实例。该类允许我们初始化一个打印任务并开始打印任务的生命周期。下面的代码展示了如何获得打印管理器并开始打印进程。

```
private void doPrint() {
    // Get a PrintManager instance
    PrintManager printManager = (PrintManager) getActivity()
            .getSystemService(Context.PRINT_SERVICE);

    // Set job name, which will be displayed in the print queue
    String jobName = getActivity().getString(R.string.app_name) + " Document";

    // Start a print job, passing in a PrintDocumentAdapter implementation
    // to handle the generation of a print document
    printManager.print(jobName, new MyPrintDocumentAdapter(getActivity()),
            null); //
}
```

上面的代码展示了如何命名一个打印任务以及如何设置一个PrintDocumentAdapter类的实例，它负责处理打印生命周期的每一步。打印适配器的实现会在下一节中进行讨论。

> **Note**：print()方法的最后一个参数接收一个PrintAttributes对象。我们可以使用这个参数向打印框架进行一些打印设置，以及基于前一个打印周期的预设，从而改善用户体验。我们也可以使用这个参数对打印内容进行一些更符合实际情况的设置，比如当打印一幅照片时，设置打印的方向与照片方向一致。

创建打印适配器

打印适配器负责与Android打印框架交互并处理打印过程的每一步。这个过程需要用户在创建打印文档前选择打印机和打印选项。由于用户可以选择不同性能的打印机，不同的页面尺寸或不同的页面方向，因此这些选项可能会影响最终的打印效果。当这些选项配置好之后，打印框架会寻求适配器进行布局并生成一个打印文档，以此作为打印的前期准备。一旦用户点击了打印按钮，框架会将最终的打印文档传递给Print Provider进行打印输出。在打印过程中，用户可以选择取消打印，所以打印适配器必须监听并响应取消打印的请求。

PrintDocumentAdapter抽象类负责处理打印的生命周期，它有四个主要的回调方法。我们必须在打印适配器中实现这些方法，以此来正确地和Android打印框架进行交互：

- onStart()：一旦打印进程开始，该方法就将被调用。如果我们的应用有任何一次性的准备任务要执行，比如获取一个要打印数据的快照，那么让它们在此处执行。在你的适配器中，这个回调方法不是必须实现的。
- onLayout()：每当用户改变了影响打印输出的设置时（比如改变了页面的尺寸，或者页面的方向）该函数将会被调用，以此给我们的应用一个机会去重新计算打印页面的布局。另外，该方法必须返回打印文档包含多少页面。
- onWrite()：该方法调用后，会将打印页面渲染成一个待打印的文件。该方法可以在onLayout()方法被调用后调用一次或多次。
- onFinish()：一旦打印进程结束后，该方法将会被调用。如果我们的应用有任何一次性销毁任务要执行，让这些任务在该方法内执行。这个回调方法不是必须实现的。

下面将介绍如何实现 `onLayout()` 以及 `onWrite()` 方法，他们是打印适配器的核心功能。

> **Note**：这些适配器的回调方法会在应用的主线程上被调用。如果这些方法的实现在执行时可能需要花费大量的时间，那么应该将他们放在另一个线程里执行。例如：我们可以将布局或者写入打印文档的操作封装在一个AsyncTask对象中。

计算打印文档信息

在实现PrintDocumentAdapter类时，我们的应用必须能够指定出所创建文档的类型，计算出打印任务所需要打印的总页数，并提供打印页面的尺寸信息。在实现适配器的onLayout()方法时，我们执行这些计算，并提供与理想的输出相关的一些信息，这些信息可以在PrintDocumentInfo类中获取，包括页数和内容类型。下面的例子展示了PrintDocumentAdapter中onLayout()方法的基本实现：

```
@Override
public void onLayout(PrintAttributes oldAttributes,
                     PrintAttributes newAttributes,
                     CancellationSignal cancellationSignal,
                     LayoutResultCallback callback,
                     Bundle metadata) {
    // Create a new PdfDocument with the requested page attributes
    mPdfDocument = new PrintedPdfDocument(getActivity(), newAttributes);

    // Respond to cancellation request
    if (cancellationSignal.isCancelled() ) {
        callback.onLayoutCancelled();
        return;
    }

    // Compute the expected number of printed pages
    int pages = computePageCount(newAttributes);

    if (pages > 0) {
        // Return print information to print framework
        PrintDocumentInfo info = new PrintDocumentInfo
                .Builder("print_output.pdf")
                .setContentType(PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
                .setPageCount(pages);
                .build();
        // Content layout reflow is complete
        callback.onLayoutFinished(info, true);
    } else {
        // Otherwise report an error to the print framework
        callback.onLayoutFailed("Page count calculation failed.");
    }
}
```

onLayout()方法的执行结果有三种：完成，取消或失败（计算布局无法顺利完成时会失败）。我们必须通过调用PrintDocumentAdapter.LayoutResultCallback对象中的适当方法来指出这些结果中的一个。

> **Note**：onLayoutFinished()方法的布尔类型参数明确了这个布局内容是否和上一次打印请求相比发生了改变。恰当地设定了这个参数将避免打印框架不必要地调用onWrite()方法，缓存之前的打印文档，提升执行性能。

onLayout()的主要任务是计算打印文档的页数，并将它作为打印参数交给打印机。如何计算页数则高度依赖于应用是如何对打印页面进行布局的。下面的代码展示了页数是如何根据打印方向确定的：

```
private int computePageCount(PrintAttributes printAttributes) {
    int itemsPerPage = 4; // default item count for portrait mode

    MediaSize pageSize = printAttributes.getMediaSize();
    if (!pageSize.isPortrait()) {
        // Six items per page in landscape orientation
        itemsPerPage = 6;
    }

    // Determine number of print items
    int printItemCount = getPrintItemCount();

    return (int) Math.ceil(printItemCount / itemsPerPage);
}
```

将打印文档写入文件

当需要将打印内容输出到一个文件时，Android打印框架会调用PrintDocumentAdapter类的onWrite()方法。这个方法的参数指定了哪些页面要被写入以及要使用的输出文件。该方法的实现必须将每一个请求页的内容渲染成一个含有多个页面的PDF文件。当这个过程结束以后，你需要调用callback对象的onWriteFinished()方法。

> **Note**： Android打印框架可能会在每次调用onLayout()后，调用onWrite()方法一次甚至更多次。请务必牢记：当打印内容的布局没有变化时，可以将onLayoutFinished()方法的布尔参数设置为“false”，以此避免对打印文档进行不必要的重写操作。
>
> **Note**：onLayoutFinished()方法的布尔类型参数明确了这个布局内容是否和上一次打印请求相比发生了改变。恰当地设定了这个参数将避免打印框架不必要的调用onLayout()方法，缓存之前的打印文档，提升执行性能。

下面的代码展示了使用PrintedPdfDocument类创建了PDF文件的基本原理：

```
@Override
public void onWrite(final PageRange[] pageRanges,
                    final ParcelFileDescriptor destination,
                    final CancellationSignal cancellationSignal,
                    final WriteResultCallback callback) {
    // Iterate over each page of the document,
    // check if it's in the output range.
    for (int i = 0; i < totalPages; i++) {
        // Check to see if this page is in the output range.
        if (containsPage(pageRanges, i)) {
            // If so, add it to writtenPagesArray. writtenPagesArray.size()
            // is used to compute the next output page index.
            writtenPagesArray.append(writtenPagesArray.size(), i);
            PdfDocument.Page page = mPdfDocument.startPage(i);

            // check for cancellation
            if (cancellationSignal.isCancelled()) {
                callback.onWriteCancelled();
                mPdfDocument.close();
                mPdfDocument = null;
                return;
            }

            // Draw page content for printing
            drawPage(page);

            // Rendering is complete, so page can be finalized.
            mPdfDocument.finishPage(page);
        }
    }

    // Write PDF document to file
    try {
        mPdfDocument.writeTo(new FileOutputStream(
                destination.getFileDescriptor()));
    } catch (IOException e) {
        callback.onWriteFailed(e.toString());
        return;
    } finally {
        mPdfDocument.close();
        mPdfDocument = null;
    }
    PageRange[] writtenPages = computeWrittenPages();
    // Signal the print framework the document is complete
    callback.onWriteFinished(writtenPages);

    ...
}
```

代码中将PDF页面递交给了drawPage()方法，这个方法会在下一部分介绍。

就布局而言，onWrite()方法的执行可以有三种结果：完成，取消或者失败（内容无法被写入）。我们必须通过调用PrintDocumentAdapter.WriteResultCallback对象中的适当方法来指明这些结果中的一个。

> **Note**：渲染打印文档是一个可能耗费大量资源的操作。为了避免阻塞应用的主UI线程，我们应该考虑将页面的渲染和写操作放在另一个线程中执行，比如在AsyncTask中执行。关于更多异步任务线程的知识，可以阅读：Processes and Threads。

绘制PDF页面内容

当我们的应用进行打印时，应用必须生成一个PDF文档并将它传递给Android打印框架以进行打印。我们可以使用任何PDF生成库来协助完成这个操作。本节将展示如何使用PrintedPdfDocument类将打印内容生成为PDF页面。

PrintedPdfDocument类使用一个Canvas对象来在PDF页面上绘制元素，这一点和在activity布局上进行绘制很类似。我们可以在打印页面上使用Canvas类提供的相关绘图方法绘制页面元素。下面的代码展示了如何使用这些方法在PDF页面上绘制一些简单的元素：

```
private void drawPage(PdfDocument.Page page) {
    Canvas canvas = page.getCanvas();

    // units are in points (1/72 of an inch)
    int titleBaseLine = 72;
    int leftMargin = 54;

    Paint paint = new Paint();
    paint.setColor(Color.BLACK);
    paint.setTextSize(36);
    canvas.drawText("Test Title", leftMargin, titleBaseLine, paint);

    paint.setTextSize(11);
    canvas.drawText("Test paragraph", leftMargin, titleBaseLine + 25, paint);

    paint.setColor(Color.BLUE);
    canvas.drawRect(100, 100, 172, 172, paint);
}
```

当使用Canvas在一个PDF页面上绘图时，元素通过单位“点（point）”来指定大小，一个点相当于七十二分之一英寸。在编写程序时，请确保使用该测量单位来指定页面上的元素大小。在定位绘制的元素时，坐标系的原点（即(0,0)点）在页面的最左上角。

> **Tip**：虽然Canvas对象允许我们将打印元素放置在一个PDF文档的边缘，但许多打印机无法在纸张的边缘打印。所以当我们使用这个类构建一个打印文档时，需要考虑到那些无法打印的边缘区域。

Android图像与动画

> 编写:kesenhoo - 原文:http://developer.android.com/training/building-graphics.html

这些课程教你如何使用图形完成任务，这会使你的app在竞争中占优势。如果你想创建超越基本用户界面的漂亮的视觉体验，这些课程会帮助你做到。

高效显示Bitmap(Displaying Bitmaps Efficiently) - 官方最新已经移除的章节

如何在加载并处理bitmaps的同时保持用户界面响应，防止超出内存限制。

使用OpenGL ES显示图像(Displaying Graphics with OpenGL ES)

如何使用Android app framework绘制OpenGL图形并响应触摸。

Animating Views Using Scenes and Transitions - 待翻译

How to animate state changes in a view hierarchy using transitions.

添加动画(Adding Animations)

如何给你的用户界面添加过渡动画。

高效显示Bitmap

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/index.html

这一章节会介绍一些处理与加载Bitmap对象的常用方法，这些技术能够使得程序的UI不会被阻塞，并且可以避免程序超出内存限制。如果我们不注意这些，Bitmaps会迅速的消耗掉可用内存从而导致程序崩溃，出现下面的异常: `java.lang.OutofMemoryError: bitmap size exceeds VM budget.`

在Android应用中加载Bitmaps的操作是需要特别小心处理的，有下面几个方面的原因:

- 移动设备的系统资源有限。Android设备对于单个程序至少需要16MB的内存。Android Compatibility Definition Document (CDD), Section 3.7. Virtual Machine Compatibility 中给出了对于不同大小与密度的屏幕的最低内存需求。 应用应该在这个最低内存限制下去优化程序的效率。当然，大多数设备的都有更高的限制需求。
- Bitmap会消耗很多内存，特别是对于类似照片等内容更加丰富的图片。 例如，Galaxy Nexus的照相机能够拍摄2592x1936 pixels (5 MB)的图片。 如果bitmap的图像配置是使用ARGB_8888 (从Android 2.3开始的默认配置) ，那么加载这张照片到内存大约需要19MB(`2592*1936*4` bytes) 的空间，从而迅速消耗掉该应用的剩余内存空间。
- Android应用的UI通常会在一次操作中立即加载许多张bitmaps。 例如在ListView, GridView 与 ViewPager 等控件中通常会需要一次加载许多张bitmaps，而且需要预先加载一些没有在屏幕上显示的内容，为用户滑动的显示做准备。

参考资料

- DEMO：DisplayingBitmaps.zip
- VEDIO：Bitmap Allocation
- VEDIO：Making App Beautiful - Part 4 - Performance Tuning

章节课程

- **高效的加载大图(Loading Large Bitmaps Efficiently)**

  这节课会带领你学习如何解析很大的Bitmaps并且避免超出程序的内存限制。

- **非UI线程处理Bitmap(Processing Bitmaps Off the UI Thread)**

  处理Bitmap（裁剪，下载等操作）不能执行在主线程。这节课会带领你学习如何使用AsyncTask在后台线程对Bitmap进行处理，并解释如何处理并发带来的问题。

- 缓存Bitmaps(Caching Bitmaps)

  这节课会带领你学习如何使用内存与磁盘缓存来提升加载多张Bitmaps时的响应速度与流畅度。

- 管理Bitmap的内存使用(Managing Bitmap Memory)

  这节课会介绍如何管理Bitmap的内存占用，以此来提升程序的性能。

- 在UI上显示Bitmap(Displaying Bitmaps in Your UI)

  这节课会综合之前章节的内容，演示如何在诸如ViewPager与GridView等控件中使用后台线程与缓存加载多张Bitmaps。

高效加载大图

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/load-bitmap.html

图片有不同的形状与大小。在大多数情况下它们的实际大小都比需要呈现的尺寸大很多。例如，系统的图库应用会显示那些我们使用相机拍摄的照片，但是那些图片的分辨率通常都比设备屏幕的分辨率要高很多。

考虑到应用是在有限的内存下工作的，理想情况是我们只需要在内存中加载一个低分辨率的照片即可。为了更便于显示，这个低分辨率的照片应该是与其对应的UI控件大小相匹配的。加载一个超过屏幕分辨率的高分辨率照片不仅没有任何显而易见的好处，还会占用宝贵的内存资源，另外在快速滑动图片时容易产生额外的效率问题。

这一课会介绍如何通过加载一个缩小版本的图片，从而避免超出程序的内存限制。

读取位图的尺寸与类型(Read Bitmap Dimensions and Type)

BitmapFactory提供了一些解码（decode）的方法（decodeByteArray(), decodeFile(), decodeResource()等），用来从不同的资源中创建一个Bitmap。 我们应该根据图片的数据源来选择合适的解码方法。 这些方法在构造位图的时候会尝试分配内存，因此会容易导致 `OutOfMemory` 的异常。每一种解码方法都可以通过BitmapFactory.Options设置一些附加的标记，以此来指定解码选项。设置 inJustDecodeBounds 属性为 `true` 可以在解码的时候避免内存的分配，它会返回一个 `null` 的Bitmap，但是可以获取到 outWidth, outHeight 与 outMimeType。该技术可以允许你在构造Bitmap之前优先读图片的尺寸与类型。

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

为了避免 `java.lang.OutOfMemory` 的异常，我们需要在真正解析图片之前检查它的尺寸（除非你能确定这个数据源提供了准确无误的图片且不会导致占用过多的内存）。

加载一个按比例缩小的版本到内存中(Load a Scaled Down Version into Memory)

通过上面的步骤我们已经获取到了图片的尺寸，这些数据可以用来帮助我们决定应该加载整个图片到内存中还是加载一个缩小的版本。有下面一些因素需要考虑：

- 评估加载完整图片所需要耗费的内存。
- 程序在加载这张图片时可能涉及到的其他内存需求。
- 呈现这张图片的控件的尺寸大小。
- 屏幕大小与当前设备的屏幕密度。

例如，如果把一个大小为1024x768像素的图片显示到大小为128x96像素的ImageView上吗，就没有必要把整张原图都加载到内存中。

为了告诉解码器去加载一个缩小版本的图片到内存中，需要在BitmapFactory.Options 中设置 inSampleSize 的值。例如, 一个分辨率为2048x1536的图片，如果设置 inSampleSize 为4，那么会产出一个大约512x384大小的Bitmap。加载这张缩小的图片仅仅使用大概0.75MB的内存，如果是加载完整尺寸的图片，那么大概需要花费12MB（前提都是Bitmap的配置是ARGB_8888）。下面有一段根据目标图片大小来计算Sample图片大小的代码示例：

```
public static int calculateInSampleSize(
            BitmapFactory.Options options, int reqWidth, int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {

        final int halfHeight = height / 2;
        final int halfWidth = width / 2;

        // Calculate the largest inSampleSize value that is a power of 2 and keeps both

        // height and width larger than the requested height and width.
        while ((halfHeight / inSampleSize) > reqHeight
                && (halfWidth / inSampleSize) > reqWidth) {
            inSampleSize *= 2;
        }
    }

    return inSampleSize;
}
```

> **Note:** 设置inSampleSize为2的幂是因为解码器最终还是会对非2的幂的数进行向下处理，获取到最靠近2的幂的数。详情参考inSampleSize的文档。

为了使用该方法，首先需要设置 inJustDecodeBounds 为 `true`, 把options的值传递过来，然后设置 inSampleSize 的值并设置 inJustDecodeBounds 为 `false` ，之后重新调用相关的解码方法。

```
public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
        int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

使用上面这个方法可以简单地加载一张任意大小的图片。如下面的代码样例显示了一个接近100x100像素的缩略图：

```
mImageView.setImageBitmap(
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));
```

我们可以通过替换合适的BitmapFactory.decode* 方法来实现一个类似的方法，从其他的数据源解析Bitmap。

非UI线程处理Bitmap

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/process-bitmap.html

在上一课中介绍了一系列的BitmapFactory.decode*方法，当图片来源是网络或者是存储卡时（或者是任何不在内存中的形式），这些方法都不应该在UI 线程中执行。因为在上述情况下加载数据时，其执行时间是不可估计的，它依赖于许多因素（从网络或者存储卡读取数据的速度，图片的大小，CPU的速度等）。如果其中任何一个子操作阻塞了UI线程，系统都会容易出现应用无响应的错误。

这一节课会介绍如何使用AsyncTask在后台线程中处理Bitmap并且演示如何处理并发（concurrency）的问题。

使用AsyncTask(Use a AsyncTask)

AsyncTask 类提供了一个在后台线程执行一些操作的简单方法，它还可以把后台的执行结果呈现到UI线程中。下面是一个加载大图的示例：

```
class BitmapWorkerTask extends AsyncTask {
    private final WeakReference imageViewReference;
    private int data = 0;

    public BitmapWorkerTask(ImageView imageView) {
        // Use a WeakReference to ensure the ImageView can be garbage collected
        imageViewReference = new WeakReference(imageView);
    }

    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        data = params[0];
        return decodeSampledBitmapFromResource(getResources(), data, 100, 100));
    }

    // Once complete, see if ImageView is still around and set bitmap.
    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

为ImageView使用WeakReference确保了AsyncTask所引用的资源可以被垃圾回收器回收。由于当任务结束时不能确保ImageView仍然存在，因此我们必须在 `onPostExecute()` 里面对引用进行检查。该ImageView在有些情况下可能已经不存在了，例如，在任务结束之前用户使用了回退操作，或者是配置发生了改变（如旋转屏幕等）。

开始异步加载位图，只需要创建一个新的任务并执行它即可:

```
public void loadBitmap(int resId, ImageView imageView) {
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);
    task.execute(resId);
}
```

处理并发问题(Handle Concurrency)

通常类似ListView与GridView等视图控件在使用上面演示的AsyncTask 方法时，会同时带来并发的问题。首先为了更高的效率，ListView与GridView的子Item视图会在用户滑动屏幕时被循环使用。如果每一个子视图都触发一个AsyncTask，那么就无法确保关联的视图在结束任务

时，分配的视图已经进入循环队列中，给另外一个子视图进行重用。而且， 无法确保所有的异步任务的完成顺序和他们本身的启动顺序保持一致。

Multithreading for Performance 这篇博文更进一步的讨论了如何处理并发问题，并且提供了一种解决方法：ImageView保存最近使用的AsyncTask的引用，这个引用可以在任务完成的时候再次读取检查。使用这种方式, 就可以对前面提到的AsyncTask进行扩展。

创建一个专用的Drawable的子类来储存任务的引用。在这种情况下，我们使用了一个BitmapDrawable，在任务执行的过程中，一个占位图片会显示在ImageView中:

```
static class AsyncDrawable extends BitmapDrawable {
    private final WeakReference bitmapWorkerTaskReference;

    public AsyncDrawable(Resources res, Bitmap bitmap,
            BitmapWorkerTask bitmapWorkerTask) {
        super(res, bitmap);
        bitmapWorkerTaskReference =
            new WeakReference(bitmapWorkerTask);
    }

    public BitmapWorkerTask getBitmapWorkerTask() {
        return bitmapWorkerTaskReference.get();
    }
}
```

在执行BitmapWorkerTask 之前，你需要创建一个AsyncDrawable并且将它绑定到目标控件ImageView中：

```
public void loadBitmap(int resId, ImageView imageView) {
    if (cancelPotentialWork(resId, imageView)) {
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
                new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);
        imageView.setImageDrawable(asyncDrawable);
        task.execute(resId);
    }
}
```

在上面的代码示例中， `cancelPotentialWork` 方法检查是否有另一个正在执行的任务与该ImageView关联了起来，如果的确是这样，它通过执行 `cancel()` 方法来取消另一个任务。在少数情况下, 新创建的任务数据可能会与已经存在的任务相吻合，这样的话就不需要进行下一步动作了。下面是 `cancelPotentialWork` 方法的实现 。

```
public static boolean cancelPotentialWork(int data, ImageView imageView) {
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

    if (bitmapWorkerTask != null) {
        final int bitmapData = bitmapWorkerTask.data;
        if (bitmapData == 0 || bitmapData != data) {
            // Cancel previous task
            bitmapWorkerTask.cancel(true);
        } else {
            // The same work is already in progress
            return false;
        }
    }
    // No task associated with the ImageView, or an existing task was cancelled
    return true;
}
```

在上面的代码中有一个辅助方法：`getBitmapWorkerTask()`，它被用作检索AsyncTask是否已经被分配到指定的ImageView:

```
private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
   if (imageView != null) {
       final Drawable drawable = imageView.getDrawable();
       if (drawable instanceof AsyncDrawable) {
           final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
           return asyncDrawable.getBitmapWorkerTask();
       }
    }
    return null;
}
```

最后一步是在BitmapWorkerTask的 `onPostExecute()` 方法里面做更新操作:

```
class BitmapWorkerTask extends AsyncTask {
    ...

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            final BitmapWorkerTask bitmapWorkerTask =
                    getBitmapWorkerTask(imageView);
            if (this == bitmapWorkerTask && imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

这个方法不仅仅适用于ListView与GridView控件，在那些需要循环利用子视图的控件中同样适用：只需要在设置图片到ImageView的地方调用 `loadBitmap` 方法。例如，在GridView 中实现这个方法可以在 getView()中调用。

缓存Bitmap

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/cache-bitmap.html

将单个Bitmap加载到UI是简单直接的，但是如果我们需要一次性加载大量的图片，事情则会变得复杂起来。在大多数情况下（例如在使用ListView，GridView或ViewPager时），屏幕上的图片和因滑动将要显示的图片的数量通常是没有限制的。

通过循环利用子视图可以缓解内存的使用，垃圾回收器也会释放那些不再需要使用的Bitmap。这些机制都非常好，但是为了保证一个流畅的用户体验，我们希望避免在每次屏幕滑动回来时，都要重复处理那些图片。内存与磁盘缓存通常可以起到辅助作用，允许控件可以快速地重新加载那些处理过的图片。

这一课会介绍在加载多张Bitmap时使用内存缓存与磁盘缓存来提高响应速度与UI流畅度。

使用内存缓存(Use a Memory Cache)

内存缓存以花费宝贵的程序内存为前提来快速访问位图。LruCache类（在API Level 4的Support Library中也可以找到）特别适合用来缓存Bitmaps，它使用一个强引用（strong referenced）的LinkedHashMap保存最近引用的对象，并且在缓存超出设置大小的时候剔除（evict）最近最少使用到的对象。

> **Note:** 在过去，一种比较流行的内存缓存实现方法是使用软引用（SoftReference）或弱引用（WeakReference）对Bitmap进行缓存，然而我们并不推荐这样的做法。从Android 2.3 (API Level 9)开始，垃圾回收机制变得更加频繁，这使得释放软（弱）引用的频率也随之增高，导致使用引用的效率降低很多。而且在Android 3.0 (API Level 11)之前，备份的Bitmap会存放在Native Memory中，它不是以可预知的方式被释放的，这样可能导致程序超出它的内存限制而崩溃。

为了给LruCache选择一个合适的大小，需要考虑到下面一些因素：

- 应用剩下了多少可用的内存?
- 多少张图片会同时呈现到屏幕上？有多少图片需要准备好以便马上显示到屏幕？
- 设备的屏幕大小与密度是多少？一个具有特别高密度屏幕（xhdpi）的设备，像Galaxy Nexus会比Nexus S（hdpi）需要一个更大的缓存空间来缓存同样数量的图片。
- Bitmap的尺寸与配置是多少，会花费多少内存？
- 图片被访问的频率如何？是其中一些比另外的访问更加频繁吗？如果是，那么我们可能希望在内存中保存那些最常访问的图片，或者根据访问频率给Bitmap分组，为不同的Bitmap组设置多个LruCache对象。

- 是否可以在缓存图片的质量与数量之间寻找平衡点？某些时候保存大量低质量的Bitmap会非常有用，加载更高质量图片的任务可以交给另外一个后台线程。

通常没有指定的大小或者公式能够适用于所有的情形，我们需要分析实际的使用情况后，提出一个合适的解决方案。缓存太小会导致额外的花销却没有明显的好处，缓存太大同样会导致java.lang.OutOfMemory的异常，并且使得你的程序只留下小部分的内存用来工作（缓存占用太多内存，导致其他操作会因为内存不够而抛出异常）。

下面是一个为Bitmap建立LruCache的示例：

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Get max available VM memory, exceeding this amount will throw an
    // OutOfMemory exception. Stored in kilobytes as LruCache takes an
    // int in its constructor.
    final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);

    // Use 1/8th of the available memory for this memory cache.
    final int cacheSize = maxMemory / 8;

    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // The cache size will be measured in kilobytes rather than
            // number of items.
            return bitmap.getByteCount() / 1024;
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

> **Note:**在上面的例子中, 有1/8的内存空间被用作缓存。 这意味着在常见的设备上（hdpi），最少大概有4MB的缓存空间（32/8）。如果一个填满图片的GridView控件放置在800x480像素的手机屏幕上，大概会花费1.5MB的缓存空间（800x480x4 bytes），因此缓存的容量大概可以缓存2.5页的图片内容。

当加载Bitmap显示到ImageView 之前，会先从LruCache 中检查是否存在这个Bitmap。如果确实存在，它会立即被用来显示到ImageView上，如果没有找到，会触发一个后台线程去处理显示该Bitmap任务。

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);

    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        mImageView.setImageBitmap(bitmap);
    } else {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }
}
```

上面的程序中 `BitmapWorkerTask` 需要把解析好的Bitmap添加到内存缓存中：

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final Bitmap bitmap = decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100));
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);
        return bitmap;
    }
    ...
}
```

使用磁盘缓存(Use a Disk Cache)

内存缓存能够提高访问最近用过的Bitmap的速度，但是我们无法保证最近访问过的Bitmap都能够保存在缓存中。像类似GridView等需要大量数据填充的控件很容易就会用尽整个内存缓存。另外，我们的应用可能会被类似打电话等行为而暂停并退到后台，因为后台应用可能会被杀死，那么内存缓存就会被销毁，里面的Bitmap也就不存在了。一旦用户恢复应用的状态，那么应用就需要重新处理那些图片。

磁盘缓存可以用来保存那些已经处理过的Bitmap，它还可以减少那些不再内存缓存中的Bitmap的加载次数。当然从磁盘读取图片会比从内存要慢，而且由于磁盘读取操作时间是不可预期的，读取操作需要在后台线程中处理。

> **Note:**如果图片会被更频繁的访问，使用ContentProvider或许会更加合适，比如在图库应用中。

这一节的范例代码中使用了一个从Android源码中剥离出来的 `DiskLruCache` 。改进过的范例代码在已有内存缓存的基础上增加磁盘缓存的功能。

```
private DiskLruCache mDiskLruCache;
private final Object mDiskCacheLock = new Object();
private boolean mDiskCacheStarting = true;
private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB
private static final String DISK_CACHE_SUBDIR = "thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Initialize memory cache
    ...
    // Initialize disk cache on background thread
    File cacheDir = getDiskCacheDir(this, DISK_CACHE_SUBDIR);
    new InitDiskCacheTask().execute(cacheDir);
    ...
}

class InitDiskCacheTask extends AsyncTask<File, Void, Void> {
    @Override
    protected Void doInBackground(File... params) {
        synchronized (mDiskCacheLock) {
            File cacheDir = params[0];
            mDiskLruCache = DiskLruCache.open(cacheDir, DISK_CACHE_SIZE);
            mDiskCacheStarting = false; // Finished initialization
            mDiskCacheLock.notifyAll(); // Wake any waiting threads
        }
        return null;
    }
}

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);

        // Check disk cache in background thread
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);

        if (bitmap == null) { // Not found in disk cache
            // Process as normal
            final Bitmap bitmap = decodeSampledBitmapFromResource(
                    getResources(), params[0], 100, 100));
        }
```

```
        // Add final bitmap to caches
        addBitmapToCache(imageKey, bitmap);

        return bitmap;
    }
    ...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // Add to memory cache as before
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    // Also add to disk cache
    synchronized (mDiskCacheLock) {
        if (mDiskLruCache != null && mDiskLruCache.get(key) == null) {
            mDiskLruCache.put(key, bitmap);
        }
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    synchronized (mDiskCacheLock) {
        // Wait while disk cache is started from background thread
        while (mDiskCacheStarting) {
            try {
                mDiskCacheLock.wait();
            } catch (InterruptedException e) {}
        }
        if (mDiskLruCache != null) {
            return mDiskLruCache.get(key);
        }
    }
    return null;
}

// Creates a unique subdirectory of the designated app cache directory. Tries to use e
xternal
// but if not mounted, falls back on internal storage.
public static File getDiskCacheDir(Context context, String uniqueName) {
    // Check if media is mounted or storage is built-in, if so, try and use external c
ache dir
    // otherwise use internal cache dir
    final String cachePath =
            Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState()) ||
                    !isExternalStorageRemovable() ? getExternalCacheDir(context).getPa
th() :
                            context.getCacheDir().getPath();

    return new File(cachePath + File.separator + uniqueName);
}
```

> **Note**:因为初始化磁盘缓存涉及到I/O操作，所以它不应该在主线程中进行。但是这也意味着在初始化完成之前缓存可以被访问。为了解决这个问题，在上面的实现中，有一个锁对象（lock object）来确保在磁盘缓存完成初始化之前，应用无法对它进行读取。

内存缓存的检查是可以在UI线程中进行的，磁盘缓存的检查需要在后台线程中处理。磁盘操作永远都不应该在UI线程中发生。当图片处理完成后，Bitmap需要添加到内存缓存与磁盘缓存中，方便之后的使用。

处理配置改变(Handle Configuration Changes)

如果运行时设备配置信息发生改变，例如屏幕方向的改变会导致Android中当前显示的Activity先被销毁然后重启。（关于这一方面的更多信息，请参考Handling Runtime Changes）。我们需要在配置改变时避免重新处理所有的图片，这样才能提供给用户一个良好的平滑过度的体验。

幸运的是，在前面介绍使用内存缓存的部分，我们已经知道了如何建立内存缓存。这个缓存可以通过调用setRetainInstance(true))保留一个Fragment实例的方法把缓存传递给新的Activity。在这个Activity被重新创建之后，这个保留的Fragment会被重新附着上。这样你就可以访问缓存对象了，从缓存中获取到图片信息并快速的重新显示到ImageView上。

下面是配置改变时使用Fragment来保留LruCache的代码示例：

```java
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    RetainFragment retainFragment =
            RetainFragment.findOrCreateRetainFragment(getFragmentManager());
    mMemoryCache = retainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
            ... // Initialize cache here as usual
        }
        retainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache<String, Bitmap> mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
            fm.beginTransaction().add(fragment, TAG).commit();
        }
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
    }
}
```

为了测试上面的效果，可以尝试在保留Fragment与没有这样做的情况下旋转屏幕。我们会发现当保留缓存时，从内存缓存中重新绘制几乎没有延迟的现象。 内存缓存中没有的图片可能存储在磁盘缓存中。如果两个缓存中都没有，则图像会像平时正常流程一样被处理。

管理Bitmap的内存使用

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/manage-memory.html

这节课将作为缓存Bitmaps课程的进一步延伸。为了优化垃圾回收机制与Bitmap的重用，我们还有一些特定的事情可以做。 同时根据Android的不同版本，推荐的策略会有所差异。DisplayingBitmaps的示例程序会演示如何设计我们的程序，使得它能够在不同的Android平台上高效地运行.

为了给这节课奠定基础，我们首先要知道Android管理Bitmap内存使用的演变进程:

- 在Android 2.2 (API level 8)以及之前，当垃圾回收发生时，应用的线程是会被暂停的，这会导致一个延迟滞后，并降低系统效率。 从**Android 2.3**开始，添加了并发垃圾回收的机制， 这意味着在一个**Bitmap**不再被引用之后，它所占用的内存会被立即回收。
- 在Android 2.3.3 (API level 10)以及之前, 一个Bitmap的像素级数据（pixel data）是存放在Native内存空间中的。 这些数据与Bitmap本身是隔离的，Bitmap本身被存放在Dalvik堆中。我们无法预测在Native内存中的像素级数据何时会被释放，这意味着程序容易超过它的内存限制并且崩溃。 自**Android 3.0 (API Level 11)**开始， 像素级数据则是与**Bitmap**本身一起存放在**Dalvik**堆中。

下面会介绍如何在不同的Android版本上优化Bitmap内存使用。

管理Android 2.3.3及以下版本的内存使用

在Android 2.3.3 (API level 10) 以及更低版本上，推荐使用recycle()方法。 如果在应用中显示了大量的Bitmap数据，我们很可能会遇到OutOfMemoryError的错误。 recycle()方法可以使得程序更快的释放内存。

> **Caution**：只有当我们确定这个Bitmap不再需要用到的时候才应该使用recycle()。在执行recycle()方法之后，如果尝试绘制这个Bitmap， 我们将得到 `"Canvas: trying to use a recycled bitmap"` 的错误提示。

下面的代码片段演示了使用 `recycle()` 的例子。它使用了引用计数的方法（ `mDisplayRefCount` 与 `mCacheRefCount` ）来追踪一个Bitmap目前是否有被显示或者是在缓存中。并且在下面列举的条件满足时，回收Bitmap：

- `mDisplayRefCount` 与 `mCacheRefCount` 的引用计数均为 0；
- bitmap不为 `null` , 并且它还没有被回收。

```
private int mCacheRefCount = 0;
private int mDisplayRefCount = 0;
...
// Notify the drawable that the displayed state has changed.
// Keep a count to determine when the drawable is no longer displayed.
public void setIsDisplayed(boolean isDisplayed) {
    synchronized (this) {
        if (isDisplayed) {
            mDisplayRefCount++;
            mHasBeenDisplayed = true;
        } else {
            mDisplayRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

// Notify the drawable that the cache state has changed.
// Keep a count to determine when the drawable is no longer being cached.
public void setIsCached(boolean isCached) {
    synchronized (this) {
        if (isCached) {
            mCacheRefCount++;
        } else {
            mCacheRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

private synchronized void checkState() {
    // If the drawable cache and display ref counts = 0, and this drawable
    // has been displayed, then recycle.
    if (mCacheRefCount <= 0 && mDisplayRefCount <= 0 && mHasBeenDisplayed
            && hasValidBitmap()) {
        getBitmap().recycle();
    }
}

private synchronized boolean hasValidBitmap() {
    Bitmap bitmap = getBitmap();
    return bitmap != null && !bitmap.isRecycled();
}
```

管理Android 3.0及其以上版本的内存

从Android 3.0 (API Level 11)开始，引进了BitmapFactory.Options.inBitmap字段。 如果使用了这个设置字段，decode方法会在加载Bitmap数据的时候去重用已经存在的Bitmap。这意味着Bitmap的内存是被重新利用的，这样可以提升性能，并且减少了内存的分配与回收。然而，使用inBitmap有一些限制，特别是在Android 4.4 (API level 19)之前，只有同等大小的位图才可以被重用。详情请查看inBitmap文档。

保存Bitmap供以后使用

下面演示了如何将一个已经存在的Bitmap存放起来以便后续使用。当一个应用运行在Android 3.0或者更高的平台上并且Bitmap从LruCache中移除时，Bitmap的一个软引用会被存放在Hashset中，这样便于之后可能被inBitmap重用：

```
Set<SoftReference<Bitmap>> mReusableBitmaps;
private LruCache<String, BitmapDrawable> mMemoryCache;

// If you're running on Honeycomb or newer, create a
// synchronized HashSet of references to reusable bitmaps.
if (Utils.hasHoneycomb()) {
    mReusableBitmaps =
            Collections.synchronizedSet(new HashSet<SoftReference<Bitmap>>());
}

mMemoryCache = new LruCache<String, BitmapDrawable>(mCacheParams.memCacheSize) {

    // Notify the removed entry that is no longer being cached.
    @Override
    protected void entryRemoved(boolean evicted, String key,
            BitmapDrawable oldValue, BitmapDrawable newValue) {
        if (RecyclingBitmapDrawable.class.isInstance(oldValue)) {
            // The removed entry is a recycling drawable, so notify it
            // that it has been removed from the memory cache.
            ((RecyclingBitmapDrawable) oldValue).setIsCached(false);
        } else {
            // The removed entry is a standard BitmapDrawable.
            if (Utils.hasHoneycomb()) {
                // We're running on Honeycomb or later, so add the bitmap
                // to a SoftReference set for possible use with inBitmap later.
                mReusableBitmaps.add
                        (new SoftReference<Bitmap>(oldValue.getBitmap()));
            }
        }
    }
....
}
```

使用已经存在的Bitmap

在运行的程序中，decode方法会检查看是否存在可重用的Bitmap。 例如:

```
public static Bitmap decodeSampledBitmapFromFile(String filename,
        int reqWidth, int reqHeight, ImageCache cache) {

    final BitmapFactory.Options options = new BitmapFactory.Options();
    ...
    BitmapFactory.decodeFile(filename, options);
    ...

    // If we're running on Honeycomb or newer, try to use inBitmap.
    if (Utils.hasHoneycomb()) {
        addInBitmapOptions(options, cache);
    }
    ...
    return BitmapFactory.decodeFile(filename, options);
}
```

下面的代码是上述代码片段中， `addInBitmapOptions()` 方法的具体实现。 它会为inBitmap查找一个已经存在的Bitmap，并将它设置为inBitmap的值。 注意这个方法只有在找到合适且可重用的Bitmap时才会赋值给inBitmap（我们需要在赋值之前进行检查）：

```
private static void addInBitmapOptions(BitmapFactory.Options options,
        ImageCache cache) {
    // inBitmap only works with mutable bitmaps, so force the decoder to
    // return mutable bitmaps.
    options.inMutable = true;

    if (cache != null) {
        // Try to find a bitmap to use for inBitmap.
        Bitmap inBitmap = cache.getBitmapFromReusableSet(options);

        if (inBitmap != null) {
            // If a suitable bitmap has been found, set it as the value of
            // inBitmap.
            options.inBitmap = inBitmap;
        }
    }
}

// This method iterates through the reusable bitmaps, looking for one
// to use for inBitmap:
protected Bitmap getBitmapFromReusableSet(BitmapFactory.Options options) {
        Bitmap bitmap = null;

    if (mReusableBitmaps != null && !mReusableBitmaps.isEmpty()) {
        synchronized (mReusableBitmaps) {
            final Iterator<SoftReference<Bitmap>> iterator
                    = mReusableBitmaps.iterator();
            Bitmap item;

            while (iterator.hasNext()) {
                item = iterator.next().get();

                if (null != item && item.isMutable()) {
                    // Check to see it the item can be used for inBitmap.
                    if (canUseForInBitmap(item, options)) {
                        bitmap = item;

                        // Remove from reusable set so it can't be used again.
                        iterator.remove();
                        break;
                    }
                } else {
                    // Remove from the set if the reference has been cleared.
                    iterator.remove();
                }
            }
        }
    }
    return bitmap;
}
```

最后，下面这个方法判断候选Bitmap是否满足inBitmap的大小条件:

```
static boolean canUseForInBitmap(
        Bitmap candidate, BitmapFactory.Options targetOptions) {

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
        // From Android 4.4 (KitKat) onward we can re-use if the byte size of
        // the new bitmap is smaller than the reusable bitmap candidate
        // allocation byte count.
        int width = targetOptions.outWidth / targetOptions.inSampleSize;
        int height = targetOptions.outHeight / targetOptions.inSampleSize;
        int byteCount = width * height * getBytesPerPixel(candidate.getConfig());
        return byteCount <= candidate.getAllocationByteCount();
    }

    // On earlier versions, the dimensions must match exactly and the inSampleSize mus
t be 1
    return candidate.getWidth() == targetOptions.outWidth
            && candidate.getHeight() == targetOptions.outHeight
            && targetOptions.inSampleSize == 1;
}

/**
 * A helper function to return the byte usage per pixel of a bitmap based on its confi
guration.
 */
static int getBytesPerPixel(Config config) {
    if (config == Config.ARGB_8888) {
        return 4;
    } else if (config == Config.RGB_565) {
        return 2;
    } else if (config == Config.ARGB_4444) {
        return 2;
    } else if (config == Config.ALPHA_8) {
        return 1;
    }
    return 1;
}
```

在UI上显示Bitmap

> 编写:kesenhoo - 原文:http://developer.android.com/training/displaying-bitmaps/display-bitmap.html

这一课会演示如何运用前面几节课的内容，使用后台线程与缓存机制将图片加载到ViewPager与GridView控件，并且学习处理并发与配置改变问题。

实现加载图片到ViewPager

Swipe View Pattern是一个使用滑动来切换显示不同详情页面的设计模型。（关于这种效果请先参看Android Design: Swipe Views）。我们可以通过PagerAdapter与ViewPager控件来实现这个效果。不过，一个更加合适的Adapter是PagerAdapter的一个子类，叫做FragmentStatePagerAdapter：它可以在某个ViewPager中的子视图切换出屏幕时自动销毁与保存Fragments的状态。这样能够保持更少的内存消耗。

> **Note:** 如果只有为数不多的图片并且确保不会超出程序内存限制，那么使用PagerAdapter或 FragmentPagerAdapter会更加合适。

下面是一个使用ViewPager与ImageView作为子视图的示例。主Activity包含有ViewPager和Adapter。

```
public class ImageDetailActivity extends FragmentActivity {
    public static final String EXTRA_IMAGE = "extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_im
age_3,
            R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_im
age_6,
            R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_im
age_9};

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_detail_pager); // Contains just a ViewPager

        mAdapter = new ImagePagerAdapter(getSupportFragmentManager(), imageResIds.leng
th);
        mPager = (ViewPager) findViewById(R.id.pager);
        mPager.setAdapter(mAdapter);
    }

    public static class ImagePagerAdapter extends FragmentStatePagerAdapter {
        private final int mSize;

        public ImagePagerAdapter(FragmentManager fm, int size) {
            super(fm);
            mSize = size;
        }

        @Override
        public int getCount() {
            return mSize;
        }

        @Override
        public Fragment getItem(int position) {
            return ImageDetailFragment.newInstance(position);
        }
    }
}
```

Fragment里面包含了ImageView控件:

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA = "resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
    public ImageDetailFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mImageNum = getArguments() != null ? getArguments().getInt(IMAGE_DATA_EXTRA) :
 -1;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        // image_detail_fragment.xml contains just an ImageView
        final View v = inflater.inflate(R.layout.image_detail_fragment, container, fal
se);
        mImageView = (ImageView) v.findViewById(R.id.imageView);
        return v;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final int resId = ImageDetailActivity.imageResIds[mImageNum];
        mImageView.setImageResource(resId); // Load image into ImageView
    }
}
```

希望你有发现上面示例存在的问题：在UI线程中读取图片可能会导致应用无响应。因此使用在第二课中学习的AsyncTask会更好。

```
public class ImageDetailActivity extends FragmentActivity {
    ...

    public void loadBitmap(int resId, ImageView imageView) {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (ImageDetailActivity.class.isInstance(getActivity())) {
            final int resId = ImageDetailActivity.imageResIds[mImageNum];
            // Call out to ImageDetailActivity to load the bitmap in a background thre
ad
            ((ImageDetailActivity) getActivity()).loadBitmap(resId, mImageView);
        }
    }
}
```

在BitmapWorkerTask中做一些例如重设图片大小，从网络拉取图片的任务，可以确保不会阻塞UI线程。如果后台线程不仅仅是一个简单的加载操作，增加一个内存缓存或者磁盘缓存会比较好（请参考第三课：缓存Bitmap），下面是一些为了内存缓存而附加的内容：

```
public class ImageDetailActivity extends FragmentActivity {
    ...
    private LruCache mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        // initialize LruCache as per Use a Memory Cache section
    }

    public void loadBitmap(int resId, ImageView imageView) {
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            mImageView.setImageBitmap(bitmap);
        } else {
            mImageView.setImageResource(R.drawable.image_placeholder);
            BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }

    ... // include updated BitmapWorkerTask from Use a Memory Cache section
}
```

把前面学习到的所有技巧合并起来，我们将得到一个响应性良好的ViewPager实现：它拥有最小的加载延迟，同时可以根据实际需求执行不同的后台处理任务。

实现加载图片到GridView

Grid List Building Block是一种有效显示大量图片的方式。它能够一次显示许多图片，同时即将被显示的图片会处于准备显示的状态。如果我们想要实现这种效果，必须确保UI是流畅的，能够控制内存使用，并且正确处理并发问题（因为GridView会循环使用子视图)。

下面是一个典型的使用场景，在Fragment里面内置GridView，其中GridView的子视图是ImageView：

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickList
ener {
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_im
age_3,
            R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_im
```

```
age_6,
            R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_im
age_9};

    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
            LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        final View v = inflater.inflate(R.layout.image_grid_fragment, container, false
);
        final GridView mGridView = (GridView) v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView parent, View v, int position, long id) {
        final Intent i = new Intent(getActivity(), ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE, position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;

        public ImageAdapter(Context context) {
            super();
            mContext = context;
        }

        @Override
        public int getCount() {
            return imageResIds.length;
        }

        @Override
        public Object getItem(int position) {
            return imageResIds[position];
        }

        @Override
        public long getItemId(int position) {
            return position;
        }
```

```
        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ImageView imageView;
            if (convertView == null) { // if it's not recycled, initialize some attrib
utes
                imageView = new ImageView(mContext);
                imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
                imageView.setLayoutParams(new GridView.LayoutParams(
                        LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT));
            } else {
                imageView = (ImageView) convertView;
            }
        //请注意下面的代码
        imageView.setImageResource(imageResIds[position]); // Load image into ImageView

        return imageView;
    }
}
```

这里同样有一个问题，上面的代码实现中，犯了把图片加载放在UI线程进行处理的错误。如果只是加载一些很小的图片，或者是经过Android系统缩放并缓存过的图片，上面的代码在运行时不会有太大问题，但是如果加载的图片稍微复杂耗时一点，这都会导致你的UI卡顿甚至应用无响应。

与前面加载图片到ViewPager一样，如果 `setImageResource` 的操作会比较耗时，也有可能会阻塞UI线程。不过我们可以使用类似前面异步处理图片与增加缓存的方法来解决这个问题。然而，我们还需要考虑GridView的循环机制所带来的并发问题。为了处理这个问题，可以参考前面的课程 。下面是一个更新过后的解决方案：

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickList
ener {
    ...

    private class ImageAdapter extends BaseAdapter {
        ...

        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ...
            loadBitmap(imageResIds[position], imageView)
            return imageView;
        }
    }

    public void loadBitmap(int resId, ImageView imageView) {
        if (cancelPotentialWork(resId, imageView)) {
            final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
            final AsyncDrawable asyncDrawable =
```

```
                    new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);
            imageView.setImageDrawable(asyncDrawable);
            task.execute(resId);
        }
    }

    static class AsyncDrawable extends BitmapDrawable {
        private final WeakReference bitmapWorkerTaskReference;

        public AsyncDrawable(Resources res, Bitmap bitmap,
                BitmapWorkerTask bitmapWorkerTask) {
            super(res, bitmap);
            bitmapWorkerTaskReference =
                new WeakReference(bitmapWorkerTask);
        }

        public BitmapWorkerTask getBitmapWorkerTask() {
            return bitmapWorkerTaskReference.get();
        }
    }

    public static boolean cancelPotentialWork(int data, ImageView imageView) {
        final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

        if (bitmapWorkerTask != null) {
            final int bitmapData = bitmapWorkerTask.data;
            if (bitmapData != data) {
                // Cancel previous task
                bitmapWorkerTask.cancel(true);
            } else {
                // The same work is already in progress
                return false;
            }
        }
        // No task associated with the ImageView, or an existing task was cancelled
        return true;
    }

    private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
       if (imageView != null) {
           final Drawable drawable = imageView.getDrawable();
           if (drawable instanceof AsyncDrawable) {
               final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
               return asyncDrawable.getBitmapWorkerTask();
           }
        }
        return null;
    }

    ... // include updated BitmapWorkerTask class
```

> **Note:**对于ListView同样可以套用上面的方法。

上面的方法提供了足够的弹性，使得我们可以做从网络下载图片，并对大尺寸大的数码照片做缩放等操作而不至于阻塞UI线程。

使用OpenGL ES显示图像

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/index.html

Android framework 提供了大量的标准视图工具，用来创建吸引人的，功能丰富的图形界面。然而，如果我们希望应用能够对屏幕上所绘制的内容进行更多的控制，或者是希望绘制3D图像，那么我们就需要一个不同的工具了。由 Android framework 提供的 OpenGL ES 接口给予我们一组可以显示高级动画和图形的工具集，它能够完成超越我们想象力的复杂多变的图形绘制。同时，这些绘制操作在绝大多数的 Android 设备上，都能够利用设备自身装载的图形处理单元（GPU）为其提供更好的性能。

这系列课程将展示如何使用 OpenGL 构建应用的基础知识，包括配置启动，绘制对象，移动图形元素以及响应点击事件。

这系列课程所涉及的样例代码使用的是 OpenGL ES 2.0 接口，这是当前Android设备所推荐的接口版本。关于更多OpenGL ES的版本信息，可以阅读：OpenGL开发手册。

> **Note**：注意不要把OpenGL ES 1.x版本的接口和OpenGL ES 2.0的接口混合调用。这两种版本的接口不是通用的。如果尝试混用它们可能会让你感到无奈和沮丧。

Sample Code

OpenGLES.zip

Lessons

- **配置OpenGL ES的环境(Building an OpenGL ES Environment)**

  学习如何配置一个可以绘制 OpenGL 图形的 Android 应用。

- **定义形状(Defining Shapes)**

  学习如何定义形状，以及为何需要了解面（Faces）和卷绕（Winding）这两个概念的原因。

- **绘制形状(Drawing Shapes)**

  学习如何在应用中利用OpenGL绘制形状。

- **运用投影与相机视角(Applying Projection and Camera Views)**

  学习如何通过投影和相机视角，获取图形对象的一个新的透视效果。

- 添加移动**(Adding Motion)**

  学习如何对一个OpenGL图形对象添加基本的运动效果。

- 响应触摸事件**(Responding to Touch Events)**

  学习如何与OpenGL图形进行基本的交互。

建立 OpenGL ES 的环境(Building an OpenGL ES Environment)

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/environment.html

要在应用中使用 OpenGL ES 绘制图像，我们必须为它们创建一个 View 容器。一种比较直接的方法是实现 GLSurfaceView 类和 GLSurfaceView.Renderer 类。其中，GLSurfaceView 是一个View容器，它用来存放使用 OpenGL 绘制的图形，而 GLSurfaceView.Renderer 则用来控制在该 View 中绘制的内容。关于这两个类的更多信息，你可以阅读：OpenGL ES 开发手册。

使用 GLSurfaceView 只是一种将 OpenGL ES 集成到应用中的方法之一。对于一个全屏的或者接近全屏的图形 View，使用它是一个合理的选择。开发者如果希望把 OpenGL ES 的图形集成到整个布局的一小部分里面，那么可以考虑使用 TextureView。对于喜欢自己动手实现的开发者来说，还可以通过使用 SurfaceView 搭建一个 OpenGL ES 的视图环境，但是这将会需要我们编写更多额外的代码。

在这节课中，我们将展示如何在一个简单的 Activity 程序中完成 GLSurfaceView 和 GLSurfaceView.Renderer 的完整落地实现。

在 Manifest 配置文件中声明使用 OpenGL ES (Declare OpenGL ES Use in the Manifest)

为了让应用能够使用 OpenGL ES 2.0 接口，我们必须将下列声明添加到 Manifest 配置文件当中：

```
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

如果我们的应用使用纹理压缩（Texture Compression），那么我们必须对支持的压缩格式也进行声明，确保应用仅安装在可以兼容的设备上：

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

更多关于纹理压缩的内容，可以阅读：OpenGL开发手册。

为 OpenGL ES 图形创建 Activity (Create an Activity for OpenGL ES Graphics)

Android 应用在呈现 OpenGL ES 的时候会使用 activity 作为用户界面，这和其他应用也同样会使用一个用户界面进行呈现交互的道理一样。主要的差别就是往 acitivity 布局内容上的的输入差异。在其他应用中你可能会使用 TextView，Button 和 ListView 等等，而在使用 OpenGL ES 的应用中，我们还可以添加一个 GLSurfaceView。

下面的代码展示了一个使用 GLSurfaceView 作为其主 View 的 Activity：

```
public class OpenGLES20Activity extends Activity {

    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a GLSurfaceView instance and set it
        // as the ContentView for this Activity.
        mGLView = new MyGLSurfaceView(this);
        setContentView(mGLView);
    }
}
```

> **Note**：OpenGL ES 2.0 需要 Android 2.2（API Level 8）或更高版本的系统，所以确保你的 Android 项目的 API 版本满足该要求。

构建一个GLSurfaceView对象(Build a GLSurfaceView Object)

GLSurfaceView 是一种比较特殊的View，我们可以在该 View 中绘制 OpenGL ES 图形，不过它自己并不做太多和绘制图形相关的任务。绘制对象的任务是由你在该 View 中配置的 GLSurfaceView.Renderer 所控制完成的。事实上，这个对象的代码非常简短，你可能不会希望继承它，直接创建一个未经修改的 GLSurfaceView 实例，不过请不要这么做，因为我们需要继承该类来捕捉触控事件，这方面知识会在响应触摸事件（该系列课程的最后一节课）中做进一步的介绍。

GLSurfaceView 的核心代码非常简短，如果想要快速的实现效果，我们通常可以在 Acitvity 中创建一个内部类并使用它：

```
class MyGLSurfaceView extends GLSurfaceView {

    private final MyGLRenderer mRenderer;

    public MyGLSurfaceView(Context context){
        super(context);

        // Create an OpenGL ES 2.0 context
        setEGLContextClientVersion(2);

        mRenderer = new MyGLRenderer();

        // Set the Renderer for drawing on the GLSurfaceView
        setRenderer(mRenderer);
    }
}
```

另一个对于 GLSurfaceView 实现的可选项，是将渲染模式设置为：GLSurfaceView.RENDERMODE_WHEN_DIRTY，其含义是：仅在你的绘制数据发生变化时才对视图进行绘制操作：

```
// Render the view only when there is a change in the drawing data
setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
```

如果选用这一配置选项，那么除非调用了 requestRender() ，否则 GLSurfaceView 不会被重新绘制，这样做可以让示例中的应用效率更高。

构建一个渲染类(Build a Renderer Class)

在一个使用 OpenGL ES 的应用中，一个 GLSurfaceView.Renderer 类的实现（或者我们将其称之为渲染器），正是事情变得有趣的地方。该类会控制和其相关联的 GLSurfaceView，具体而言，它会控制在 GLSurfaceView 上绘制的内容。在渲染器中，一共有三个方法会被Android系统调用，以此来明确要在 GLSurfaceView 上绘制的内容以及如何绘制：

- onSurfaceCreated()：调用一次，用来建立 View 的 OpenGL ES 环境。
- onDrawFrame()：每次重新绘制 View 时被调用。
- onSurfaceChanged()：如果 View 的几何形态发生变化时会被调用，例如当设备的屏幕方向发生改变时。

下面是一个非常基本的 OpenGL ES 渲染器的实现，它仅仅在 GLSurfaceView 中画一个黑色的背景：

```
public class MyGLRenderer implements GLSurfaceView.Renderer {

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        // Set the background frame color
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    }

    public void onDrawFrame(GL10 unused) {
        // Redraw background color
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    }

    public void onSurfaceChanged(GL10 unused, int width, int height) {
        GLES20.glViewport(0, 0, width, height);
    }
}
```

就是这样！上面的代码创建了一个简单地应用程序，它使用 OpenGL 让屏幕呈现为黑色。虽然它的代码看上去并没有做什么非常有意思的事情，但是通过创建这些类，我们已经对使用 OpenGL 绘制图形有了基本的认识和铺垫。

> **Note**：你可能想知道，自己明明使用的是 OpenGL ES 2.0 接口，为什么这些方法会有一个 GL10 的参数。这是因为这些方法的签名（Method Signature）在2.0接口中被简单地重用了，以此来保持Android框架的代码尽量简单。

如果你对OpenGL ES接口很熟悉，那么你现在就可以在你的应用中构建一个OpenGL ES的环境并绘制图形了。当然， 如果你希望获取更多的帮助来学会使用OpenGL，那么请继续学习下一节课程获取更多的知识。

定义形状(Defining Shapes)

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/shapes.html

在一个 OpenGL ES View 的上下文（Context）中定义形状，是创建你的杰作所需要的第一步。在了解关于 OpenGL ES 如何定义图形对象的基本知识之前，想通过 OpenGL ES 直接绘图可能会有些困难。

这节课将讲解 OpenGL ES 相对于 Android 设备屏幕的坐标系，定义形状和形状表面的基本知识，例如定义一个三角形和一个矩形。

定义一个三角形(Define a Triangle)

OpenGL ES 允许我们使用三维空间的坐标系来定义绘画对象。所以在我们能画三角形之前，必须先定义它的坐标。在 OpenGL 中，典型的办法是为坐标定义一个浮点型的顶点数组。为了高效起见，我们可以将坐标写入一个 ByteBuffer，它将会传入 OpenGl ES 的图形处理流程中：

```
public class Triangle {

    private FloatBuffer vertexBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float triangleCoords[] = {   // in counterclockwise order:
             0.0f,  0.622008459f, 0.0f, // top
            -0.5f, -0.311004243f, 0.0f, // bottom left
             0.5f, -0.311004243f, 0.0f  // bottom right
    };

    // Set color with red, green, blue and alpha (opacity) values
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };

    public Triangle() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
                // (number of coordinate values * 4 bytes per float)
                triangleCoords.length * 4);
        // use the device hardware's native byte order
        bb.order(ByteOrder.nativeOrder());

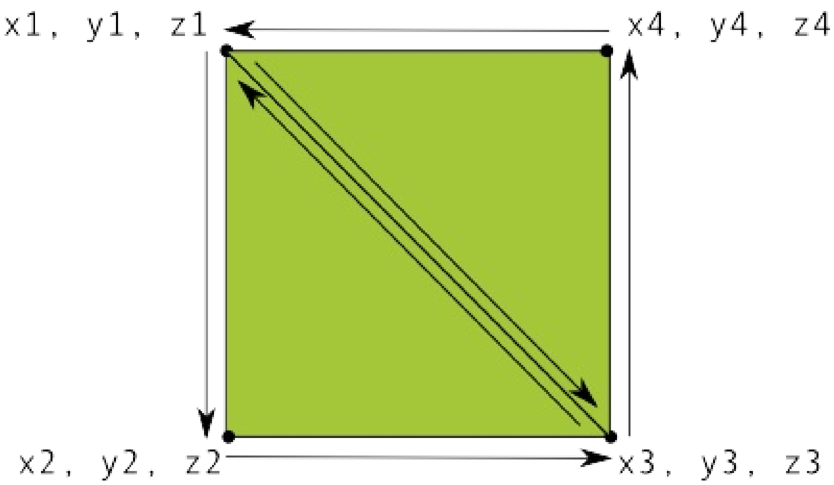
        // create a floating point buffer from the ByteBuffer
        vertexBuffer = bb.asFloatBuffer();
        // add the coordinates to the FloatBuffer
        vertexBuffer.put(triangleCoords);
        // set the buffer to read the first coordinate
        vertexBuffer.position(0);
    }
}
```

默认情况下，OpenGL ES 会假定一个坐标系，在这个坐标系中，[0, 0, 0]（分别对应X轴坐标，Y轴坐标，Z轴坐标）对应的是 GLSurfaceView 的中心。[1, 1, 0]对应的是右上角，[-1, -1, 0]对应的则是左下角。如果想要看此坐标系的插图说明，可以阅读 OpenGL ES 开发手册。

注意到这个形状的坐标是以逆时针顺序定义的。绘制的顺序非常关键，因为它定义了哪一面是形状的正面（希望绘制的一面），以及背面（使用OpenGL ES的Cull Face功能可以让背面不要绘制）。更多关于该方面的信息，可以阅读OpenGL ES开发手册。

定义一个矩形(Define a Square)

在 OpenGL 中定义三角形非常简单，那么你是否想要来点更复杂的呢？比如，定义一个矩形？有很多方法可以用来定义矩形，不过在 OpenGL ES 中最典型的办法是使用两个三角形拼接在一起：



再一次地，我们需要逆时针地为三角形顶点定义坐标来表现这个图形，并将值放入一个ByteBuffer中。为了避免由两个三角形重合的那条边的顶点被重复定义，可以使用一个绘制列表来告诉OpenGL ES图形处理流程应该如何画这些顶点。下面是代码样例：

```
public class Square {

    private FloatBuffer vertexBuffer;
    private ShortBuffer drawListBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float squareCoords[] = {
            -0.5f,  0.5f, 0.0f,   // top left
            -0.5f, -0.5f, 0.0f,   // bottom left
             0.5f, -0.5f, 0.0f,   // bottom right
             0.5f,  0.5f, 0.0f }; // top right

    private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; // order to draw vertices

    public Square() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
        // (# of coordinate values * 4 bytes per float)
                squareCoords.length * 4);
        bb.order(ByteOrder.nativeOrder());
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(squareCoords);
        vertexBuffer.position(0);

        // initialize byte buffer for the draw list
        ByteBuffer dlb = ByteBuffer.allocateDirect(
        // (# of coordinate values * 2 bytes per short)
                drawOrder.length * 2);
        dlb.order(ByteOrder.nativeOrder());
        drawListBuffer = dlb.asShortBuffer();
        drawListBuffer.put(drawOrder);
        drawListBuffer.position(0);
    }
}
```

该样例可以看作是一个如何使用 OpenGL 创建复杂图形的启发，通常来说，我们需要使用三角形的集合来绘制对象。在下一节课中，我们将学习如何在屏幕上画这些形状。

绘制形状(Drawing Shapes)

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/draw.html

在定义了将要绘制的形状之后，你可能希望使用 OpenGL 绘制出它们。使用 OpenGL ES 2.0 绘制图形可能会比你想象当中更复杂一些，因为 API 中提供了大量对于图形渲染流程的控制。

这节课将解释如何使用 OpenGL ES 2.0 接口画出在上一节课中定义的形状。

初始化形状(Initialize Shapes)

在你开始绘画之前，你需要初始化并加载你期望绘制的图形。除非你所使用的形状结构（原始坐标）在执行过程中发生了变化，不然的话你应该在渲染器的 onSurfaceCreated() 方法中初始化它们，这样做是出于内存和执行效率的考量。

```
public class MyGLRenderer implements GLSurfaceView.Renderer {

    ...
    private Triangle mTriangle;
    private Square   mSquare;

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        ...

        // initialize a triangle
        mTriangle = new Triangle();
        // initialize a square
        mSquare = new Square();
    }
    ...
}
```

画一个形状(Draw a Shape)

使用 OpenGL ES 2.0 画一个定义好的形状需要较多代码，因为你需要提供很多图形渲染流程的细节。具体而言，你必须定义如下几项：

- 顶点着色器（Vertex Shader）：用来渲染形状顶点的 OpenGL ES 代码。
- 片段着色器（Fragment Shader）：使用颜色或纹理渲染形状表面的 OpenGL ES 代码。
- 程式（Program）：一个 OpenGL ES 对象，包含了你希望用来绘制一个或更多图形所要

用到的着色器。

你需要至少一个顶点着色器来绘制一个形状，以及一个片段着色器为该形状上色。这些着色器必须被编译然后添加到一个 OpenGL ES Program 当中，并利用它来绘制形状。下面的代码在 Triangle 类中定义了基本的着色器，我们可以利用它们绘制出一个图形：

```
public class Triangle {

    private final String vertexShaderCode =
        "attribute vec4 vPosition;" +
        "void main() {" +
        "  gl_Position = vPosition;" +
        "}";

    private final String fragmentShaderCode =
        "precision mediump float;" +
        "uniform vec4 vColor;" +
        "void main() {" +
        "  gl_FragColor = vColor;" +
        "}";

    ...
}
```

着色器包含了 OpenGL Shading Language（GLSL）代码，它必须先被编译然后才能在 OpenGL 环境中使用。要编译这些代码，需要在你的渲染器类中创建一个辅助方法：

```
public static int loadShader(int type, String shaderCode){

    // create a vertex shader type (GLES20.GL_VERTEX_SHADER)
    // or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)
    int shader = GLES20.glCreateShader(type);

    // add the source code to the shader and compile it
    GLES20.glShaderSource(shader, shaderCode);
    GLES20.glCompileShader(shader);

    return shader;
}
```

为了绘制你的图形，你必须编译着色器代码，将它们添加至一个 OpenGL ES Program 对象中，然后执行链接。在你的绘制对象的构造函数里做这些事情，这样上述步骤就只用执行一次。

> **Note**：编译 OpenGL ES 着色器及链接操作对于CPU周期和处理时间而言，消耗是巨大的，所以你应该避免重复执行这些事情。如果在执行期间不知道着色器的内容，那么你应该在构建你的应用时，确保它们只被创建了一次，并且缓存以备后续使用。

```
public class Triangle() {
    ...

    private final int mProgram;

    public Triangle() {
        ...

        int vertexShader = MyGLRenderer.loadShader(GLES20.GL_VERTEX_SHADER,
                                        vertexShaderCode);
        int fragmentShader = MyGLRenderer.loadShader(GLES20.GL_FRAGMENT_SHADER,
                                        fragmentShaderCode);

        // create empty OpenGL ES Program
        mProgram = GLES20.glCreateProgram();

        // add the vertex shader to program
        GLES20.glAttachShader(mProgram, vertexShader);

        // add the fragment shader to program
        GLES20.glAttachShader(mProgram, fragmentShader);

        // creates OpenGL ES program executables
        GLES20.glLinkProgram(mProgram);
    }
}
```

至此，你已经完全准备好添加实际的调用语句来绘制你的图形了。使用OpenGL ES绘制图形需要你定义一些变量来告诉渲染流程你需要绘制的内容以及如何绘制。既然绘制属性会根据形状的不同而发生变化，把绘制逻辑包含在形状类里面将是一个不错的主意。

创建一个 `draw()` 方法来绘制图形。下面的代码为形状的顶点着色器和形状着色器设置了位置和颜色值，然后执行绘制函数：

```
private int mPositionHandle;
private int mColorHandle;

private final int vertexCount = triangleCoords.length / COORDS_PER_VERTEX;
private final int vertexStride = COORDS_PER_VERTEX * 4; // 4 bytes per vertex

public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(mProgram);

    // get handle to vertex shader's vPosition member
    mPositionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");

    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(mPositionHandle);

    // Prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
                                 GLES20.GL_FLOAT, false,
                                 vertexStride, vertexBuffer);

    // get handle to fragment shader's vColor member
    mColorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");

    // Set color for drawing the triangle
    GLES20.glUniform4fv(mColorHandle, 1, color, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

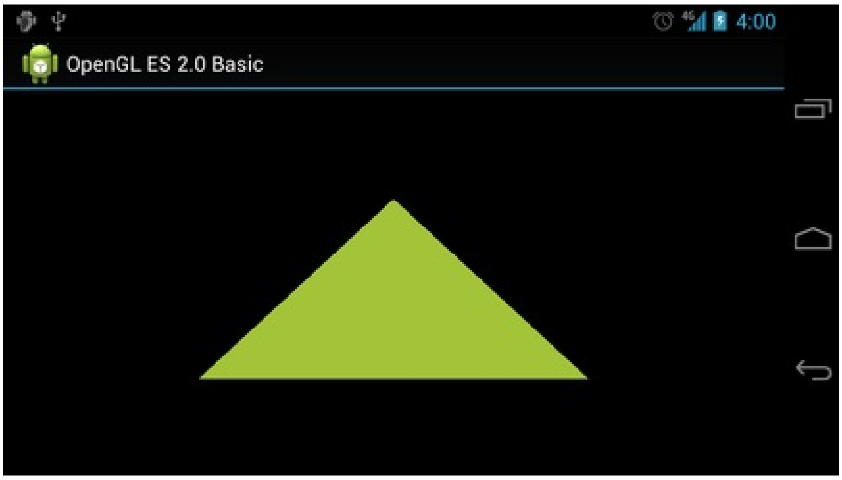
    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```

一旦你完成了上述所有代码，仅需要在你渲染器的onDrawFrame()方法中调用 `draw()` 方法就可以画出我们想要画的对象了：

```
public void onDrawFrame(GL10 unused) {
    ...

    mTriangle.draw();
}
```

当你运行这个应用时，它看上去会像是这样：



在这个代码样例中，还存在一些问题。首先，它无法给用户带来什么深刻的印象。其次，这个三角形看上去有一些扁，另外当你改变屏幕方向时，它的形状也会随之改变。发生形变的原因是因为对象的顶点没有根据显示GLSurfaceView的屏幕区域的长宽比进行修正。你可以在下一节课中使用投影（Projection）或者相机视角（Camera View）来解决这个问题。

最后，这个三角形是静止的，这看上去有些无聊。在添加移动课程当中（后续课程），你会让这个形状发生旋转，并使用一些OpenGL ES图形处理流程中更加新奇的用法。

运用投影与相机视角

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/projection.html

在OpenGL ES环境中，利用投影和相机视角可以让显示的绘图对象更加酷似于我们用肉眼看到的真实物体。该物理视角的仿真是对绘制对象坐标的进行数学变换实现的：

- 投影（**Projection**）：这个变换会基于显示它们的GLSurfaceView的长和宽，来调整绘图对象的坐标。如果没有该计算，那么用OpenGL ES绘制的对象会由于其长宽比例和View窗口比例的不一致而发生形变。一个投影变换一般仅当OpenGL View的比例在渲染器的onSurfaceChanged()方法中建立或发生变化时才被计算。关于更多OpenGL ES投影和坐标映射的知识，可以阅读Mapping Coordinates for Drawn Objects。
- 相机视角（**Camera View**）：这个变换会基于一个虚拟相机位置改变绘图对象的坐标。注意到OpenGL ES并没有定义一个实际的相机对象，取而代之的，它提供了一些辅助方法，通过对绘图对象的变换来模拟相机视角。一个相机视角变换可能仅在建立你的GLSurfaceView时计算一次，也可能根据用户的行为或者你的应用的功能进行动态调整。

这节课将解释如何创建一个投影和一个相机视角，并应用它们到GLSurfaceView中的绘制图像上。

定义一个投影

投影变换的数据会在GLSurfaceView.Renderer类的onSurfaceChanged()方法中被计算。下面的代码首先接收GLSurfaceView的高和宽，然后利用它并使用Matrix.frustumM()方法来填充一个投影变换矩阵（Projection Transformation Matrix）：

```
// mMVPMatrix is an abbreviation for "Model View Projection Matrix"
private final float[] mMVPMatrix = new float[16];
private final float[] mProjectionMatrix = new float[16];
private final float[] mViewMatrix = new float[16];

@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0, 0, width, height);

    float ratio = (float) width / height;

    // this projection matrix is applied to object coordinates
    // in the onDrawFrame() method
    Matrix.frustumM(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
```

该代码填充了一个投影矩阵：mProjectionMatrix，在下一节中，我们可以在onDrawFrame()方法中将它和一个相机视角变换结合起来。

> **Note**：在绘图对象上只应用一个投影变换会导致显示效果看上去很空旷。一般而言，我们还要实现一个相机视角，使得所有对象出现在屏幕上。

定义一个相机视角

在渲染器中添加一个相机视角变换作为绘图过程的一部分，以此完成我们的绘图对象所需变换的所有步骤。在下面的代码中，使用Matrix.setLookAtM()方法来计算相机视角变换，然后与之前计算的投影矩阵结合起来，结合后的变换矩阵传递给绘制图像：

```
@Override
public void onDrawFrame(GL10 unused) {
    ...
    // Set the camera position (View matrix)
    Matrix.setLookAtM(mViewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0, mViewMatrix, 0);

    // Draw shape
    mTriangle.draw(mMVPMatrix);
}
```

应用投影和相机变换

为了使用在之前章节中结合了的相机视角变换和投影变换，我们首先为之前在Triangle类中定义的顶点着色器添加一个Matrix变量：

```
public class Triangle {

    private final String vertexShaderCode =
        // This matrix member variable provides a hook to manipulate
        // the coordinates of the objects that use this vertex shader
        "uniform mat4 uMVPMatrix;" +
        "attribute vec4 vPosition;" +
        "void main() {" +
        // the matrix must be included as a modifier of gl_Position
        // Note that the uMVPMatrix factor *must be first* in order
        // for the matrix multiplication product to be correct.
        "  gl_Position = uMVPMatrix * vPosition;" +
        "}";

    // Use to access and set the view transformation
    private int mMVPMatrixHandle;

    ...
}
```

之后，修改图形对象的`draw()`方法，使得它接收组合后的变换矩阵，并将它应用到图形上：

```
public void draw(float[] mvpMatrix) { // pass in the calculated transformation matrix
    ...

    // get handle to shape's transformation matrix
    mMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

    // Pass the projection and view transformation to the shader
    GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```

一旦我们正确地计算并应用了投影变换和相机视角变换，我们的图形就会以正确的比例绘制出来，它看上去会像是这样:



现在，应用已经可以通过正确的比例显示图形了，下面就为图形添加一些动画效果吧！

添加移动

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/motion.html

在屏幕上绘制图形是OpenGL的一个基本特性，当然我们也可以通过其它的Android图形框架类做这些事情，包括Canvas和Drawable对象。OpenGL ES的特别之处在于，它还提供了其它的一些功能，比如在三维空间中对绘制图形进行移动和变换操作，或者通过其它独有的方法创建出引人入胜的用户体验。

在这节课中，我们会更深入地学习OpenGL ES的知识：对一个图形添加旋转动画。

旋转一个形状

使用OpenGL ES 2.0 旋转一个绘制图形是比较简单的。在渲染器中，创建另一个变换矩阵（一个旋转矩阵），并且将它和我们的投影变换矩阵以及相机视角变换矩阵结合在一起：

```
private float[] mRotationMatrix = new float[16];
public void onDrawFrame(GL10 gl) {
    float[] scratch = new float[16];

    ...

    // Create a rotation transformation for the triangle
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, angle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera view
    // Note that the mMVPMatrix factor *must be first* in order
    // for the matrix multiplication product to be correct.
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);

    // Draw triangle
    mTriangle.draw(scratch);
}
```

如果完成了这些变更以后，你的三角形还是没有旋转的话，确认一下你是否将启用GLSurfaceView.RENDERMODE_WHEN_DIRTY的这一配置所对应的代码注释掉了，有关该方面的知识会在下一节中展开。

启用连续渲染

如果严格按照这节课的样例代码走到了现在这一步，那么请确认一下是否将设置渲染模式为 `RENDERMODE_WHEN_DIRTY` 的那行代码注释了，不然的话OpenGL只会对这个形状执行一次旋转，然后就等待GLSurfaceView容器的requestRender())方法被调用后才会继续执行渲染操作。

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing data.
    // To allow the triangle to rotate automatically, this line is commented out:
    //setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}
```

除非某个对象，它的变化和用户的交互无关，不然的话一般还是建议将这个配置打开。在下一节课中的内容将会把这个注释放开，再次设定这一配置选项。

响应触摸事件

> 编写:jdneo - 原文:http://developer.android.com/training/graphics/opengl/touch.html

让对象根据预设的程序运动（如让一个三角形旋转），可以有效地引起用户的注意，但是如果希望让OpenGL ES的图形对象与用户交互呢？让我们的OpenGL ES应用可以支持触控交互的关键点在于，拓展GLSurfaceView的实现，重写onTouchEvent()方法来监听触摸事件。

这节课将会向你展示如何监听触控事件，让用户旋转一个OpenGL ES对象。

配置触摸监听器

为了让我们的OpenGL ES应用响应触控事件，我们必须实现GLSurfaceView类中的onTouchEvent()方法。下面的例子展示了如何监听MotionEvent.ACTION_MOVE事件，并将事件转换为形状旋转的角度：

```
private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
private float mPreviousX;
private float mPreviousY;

@Override
public boolean onTouchEvent(MotionEvent e) {
    // MotionEvent reports input details from the touch screen
    // and other input controls. In this case, you are only
    // interested in events where the touch position changed.

    float x = e.getX();
    float y = e.getY();

    switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:

            float dx = x - mPreviousX;
            float dy = y - mPreviousY;

            // reverse direction of rotation above the mid-line
            if (y > getHeight() / 2) {
              dx = dx * -1 ;
            }

            // reverse direction of rotation to left of the mid-line
            if (x < getWidth() / 2) {
              dy = dy * -1 ;
            }

            mRenderer.setAngle(
                    mRenderer.getAngle() +
                    ((dx + dy) * TOUCH_SCALE_FACTOR));
            requestRender();
    }

    mPreviousX = x;
    mPreviousY = y;
    return true;
}
```

注意在计算旋转角度后，该方法会调用requestRender()来告诉渲染器现在可以进行渲染了。这种办法对于这个例子来说是最有效的，因为图形并不需要重新绘制，除非有一个旋转角度的变化。当然，为了能够真正实现执行效率的提高，记得使用setRenderMode()方法以保证渲染器仅在数据发生变化时才会重新绘制图形，所以请确保这一行代码没有被注释掉：

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing data
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}
```

公开旋转角度

上述样例代码需要我们公开旋转的角度，具体来说，是在渲染器中添加一个`public`成员变量。由于渲染器代码运行在一个独立的线程中（非主UI线程），我们必须同时将该变量声明为volatile。注意下面声明该变量的代码，另外对应的get和set方法也被声明为了`public`成员函数：

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
    ...

    public volatile float mAngle;

    public float getAngle() {
        return mAngle;
    }

    public void setAngle(float angle) {
        mAngle = angle;
    }
}
```

应用旋转

为了应用触控输入所生成的旋转，注释掉创建旋转角度的代码，然后添加`mAngle`，该变量包含了触控输入所生成的角度：

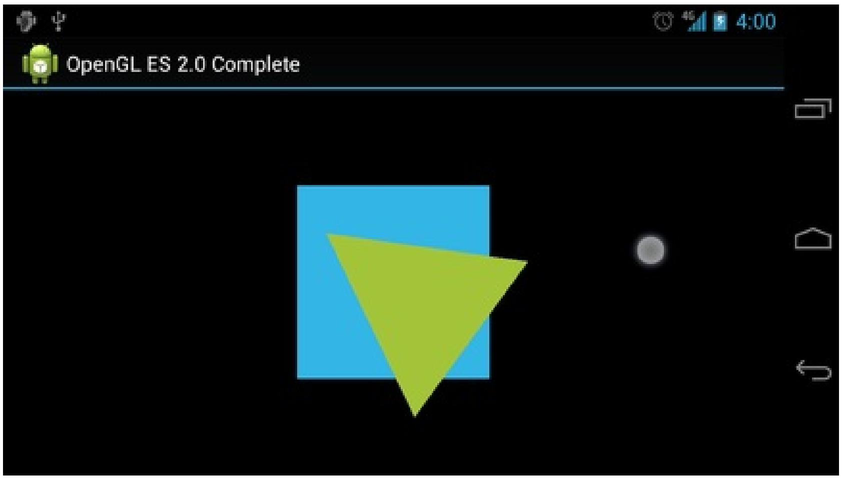
```
public void onDrawFrame(GL10 gl) {
    ...
    float[] scratch = new float[16];

    // Create a rotation for the triangle
    // long time = SystemClock.uptimeMillis() % 4000L;
    // float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera view
    // Note that the mMVPMatrix factor *must be first* in order
    // for the matrix multiplication product to be correct.
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);

    // Draw triangle
    mTriangle.draw(scratch);
}
```

当完成了上述步骤，我们就可以运行这个程序，并通过手指在屏幕上的滑动旋转三角形了：



添加动画

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/index.html

动画可以为我们的App增加精细的视觉提示，并且能改进App界面的思维模型。当界面改变其状态时（例如加载内容或新操作可用时），动画特别有帮助。另外，动画也能让我们的App外观更加优雅，为用户提供一种更好的使用体验。

但是记住：滥用动画或者在错误时机使用动画也是有害的，比如他们会造成延迟。本系列课程将会讲解如何应用常用动画类型来提升易用性。我们的目标是在不给用户用户增加烦恼的前提下提升App的“气质”。

Lessons

- View间渐变

  学习在重叠View间的淡入淡出。作为一个例子，我们将会学习如何在一个进度条与一个包含了文本内容的View之间实现淡入淡出的效果。

- 用ViewPager实现屏幕滑动

  学习怎样为水平相邻的界面提供滑动动画。

- 展示Card翻转动画

  学习怎样实现两个View之间的翻转动画。

- 缩放View

  学习怎样通过触控放大一个View。

- 布局变更动画

  学习在增加、移除或更新子View时，怎样使用内置的动画效果。

View间渐变

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/crossfade.html

渐变动画（也叫消失）通常指渐渐的淡出某个UI组件，同时同步地淡入另一个。当App想切换内容或View的情况下，这种动画很有用。渐变简短不易察觉，同时又提供从一个界面到下一个之间流畅的转换。如果在需要转换的时候没有使用任何动画效果，这会使得转换看上去感到生硬而仓促。

下面是一个利用进度指示渐变到一些文本内容的例子。



如果你想跳过这部分介绍直接查看样例，下载并运行样例App然后选择渐变例子。查看下列文件中的代码实现：

- `src/CrossfadeActivity.java`
- `layout/activity_crossfade.xml`
- `menu/activity_crossfade.xml`

创建View

创建两个我们想相互渐变的View。下面的例子创建了一个进度提示圈和可滑动文本View。

```
<FrameLayout xmlns:android="/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ScrollView xmlns:android="/apk/res/android"
        android:id="@+id/content"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView style="?android:textAppearanceMedium"
            android:lineSpacingMultiplier="1.2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/lorem_ipsum"
            android:padding="16dp" />

    </ScrollView>

    <ProgressBar android:id="@+id/loading_spinner"
        style="?android:progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

</FrameLayout>
```

设置动画

为设置动画，我们需要按照如下步骤来做：

1. 为我们想渐变的View 创建成员变量。在之后动画应用途中修改View的时候我们会需要这些引用。
2. 对于被淡入的View，设置它的visibility为 `GONE` 。这样防止view再占据布局的空间，而且也能在布局计算中将其忽略，加速处理过程。
3. 将 `config_shortAnimTime` 系统属性暂存到一个成员变量里。这个属性为动画定义了一个标准的“短”持续时间。对于细微或者快速发生的动画，这是个很理想的持续时段。也可以根据实际需求使用 `config_longAnimTime` 或 `config_mediumAnimTime` 。

下面的例子使用了前文提到的布局文件：

```
public class CrossfadeActivity extends Activity {

    private View mContentView;
    private View mLoadingView;
    private int mShortAnimationDuration;

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_crossfade);

        mContentView = findViewById(R.id.content);
        mLoadingView = findViewById(R.id.loading_spinner);

        // Initially hide the content view.
        mContentView.setVisibility(View.GONE);

        // Retrieve and cache the system's default "short" animation time.
        mShortAnimationDuration = getResources().getInteger(
                android.R.integer.config_shortAnimTime);
    }
```

渐变View

进行了上述配置之后，接下来就让我们实现渐变动画吧：

1. 对于正在淡入的View，设置它的alpha值为0并且设置visibility为 `VISIBLE` （记住他起初被设置成了 `GONE` ）。这样View就变成可见的了，但是此时它是透明的。
2. 对于正在淡入的View，把alpha值从0动态改变到1。同时，对于淡出的View，把alpha值从1动态变到0。
3. 使用 `Animator.AnimatorListener` 中的 `onAnimationEnd()` ，设置淡出View的visibility为 `GONE` 。即使alpha值为0，也要把View的visibility设置成 `GONE` 来防止 view 占据布局空间，还能把它从布局计算中忽略，加速处理过程。

详见下面的例子：

```java
private View mContentView;
private View mLoadingView;
private int mShortAnimationDuration;

...

private void crossfade() {

    // Set the content view to 0% opacity but visible, so that it is visible
    // (but fully transparent) during the animation.
    mContentView.setAlpha(0f);
    mContentView.setVisibility(View.VISIBLE);

    // Animate the content view to 100% opacity, and clear any animation
    // listener set on the view.
    mContentView.animate()
            .alpha(1f)
            .setDuration(mShortAnimationDuration)
            .setListener(null);

    // Animate the loading view to 0% opacity. After the animation ends,
    // set its visibility to GONE as an optimization step (it won't
    // participate in layout passes, etc.)
    mLoadingView.animate()
            .alpha(0f)
            .setDuration(mShortAnimationDuration)
            .setListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    mLoadingView.setVisibility(View.GONE);
                }
            });
}
```

使用ViewPager实现屏幕滑动

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/screen-slide.html

屏幕划动是在两个完整界面间的转换，它在一些UI中很常见，比如设置向导和幻灯放映。这节课将告诉你怎样通过support library提供的 `ViewPager` 实现屏幕滑动。 `ViewPager` 能自动实现屏幕滑动动画。下面展示了从一个内容界面到一下界面的屏幕滑动转换是什么样子的。



如果你想直接查看整个例子，下载并运行App样例然后选择屏幕滑动例子。查看下列文件中的代码实现：

- `src/ScreenSlidePageFragment.java`
- `src/ScreenSlideActivity.java`
- `layout/activity_screen_slide.xml`
- `layout/fragment_screen_slide_page.xml`

创建View

创建Fragment所使用的布局文件。下面的例子包含一个显示文本的TextView：

```
<!-- fragment_screen_slide_page.xml -->
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView style="?android:textAppearanceMedium"
        android:padding="16dp"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lorem_ipsum" />
</ScrollView>
```

与此同时我们还定义了一个字符串作为该Fragment的内容。

创建Fragment

创建一个 `Fragment` 子类，它从 `onCreateView()` 方法中返回之前创建的布局。无论何时如果我们需要为用户展示一个新的页面，可以在它的父Activity中创建该Fragment的实例：

```
import android.support.v4.app.Fragment;
...
public class ScreenSlidePageFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        ViewGroup rootView = (ViewGroup) inflater.inflate(
                R.layout.fragment_screen_slide_page, container, false);

        return rootView;
    }
}
```

添加ViewPager

`ViewPager` 有内建的滑动手势用来在页面间转换，并且它默认使用滑屏动画，所以我们不用自己为其创建。 `ViewPager` 使用 `PagerAdapter` 来补充新页面，所以 `PagerAdapter` 会用到你之前新建的Fragment类。

开始之前，创建一个包含 `ViewPager` 的布局：

```
<!-- activity_screen_slide.xml -->
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

创建一个Activity来做下面这些事情：

- 把ContentView设置成这个包含 `ViewPager` 的布局。
- 创建一个继承自 `FragmentStatePagerAdapter` 抽象类的类，然后实现 `getItem()` 方法来把 `ScreenSlidePageFragment` 实例作为新页面补充进来。PagerAdapter还需要实现 `getCount()` 方法，它返回 Adapter将要创建页面的总数（例如5个）。
- 把 `PagerAdapter` 和 `ViewPager` 关联起来。
- 处理Back按钮，按下变为在虚拟的Fragment栈中回退。如果用户已经在第一个页面了，则在Activity的回退栈（back stack）中回退。

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
public class ScreenSlidePagerActivity extends FragmentActivity {
    /**
     * The number of pages (wizard steps) to show in this demo.
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles animation and allows swiping horizontally to ac
cess previous
     * and next wizard steps.
     */
    private ViewPager mPager;

    /**
     * The pager adapter, which provides the pages to the view pager widget.
     */
    private PagerAdapter mPagerAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_screen_slide);

        // Instantiate a ViewPager and a PagerAdapter.
        mPager = (ViewPager) findViewById(R.id.pager);
        mPagerAdapter = new ScreenSlidePagerAdapter(getSupportFragmentManager());
        mPager.setAdapter(mPagerAdapter);
```

```
    }

    @Override
    public void onBackPressed() {
        if (mPager.getCurrentItem() == 0) {
            // If the user is currently looking at the first step, allow the system to
 handle the
            // Back button. This calls finish() on this activity and pops the back sta
ck.
            super.onBackPressed();
        } else {
            // Otherwise, select the previous step.
            mPager.setCurrentItem(mPager.getCurrentItem() - 1);
        }
    }

    /**
     * A simple pager adapter that represents 5 ScreenSlidePageFragment objects, in
     * sequence.
     */
    private class ScreenSlidePagerAdapter extends FragmentStatePagerAdapter {
        public ScreenSlidePagerAdapter(FragmentManager fm) {
            super(fm);
        }

        @Override
        public Fragment getItem(int position) {
            return new ScreenSlidePageFragment();
        }

        @Override
        public int getCount() {
            return NUM_PAGES;
        }
    }
}
```

用PageTransformer自定义动画

要展示不同于默认滑屏效果的动画，我们需要实现 `ViewPager.PageTransformer` 接口，然后把它补充到ViewPager里就行了。这个接口只暴露了一个方法， `transformPage()` 。每次界面切换，这个方法都会为每个可见页面（通常只有一个页面可见）和刚消失的相邻页面调用一次。例如，第三页可见而且用户向第四页拖动， `transformPage()` 在操作的各个阶段为第二，三，四页分别调用。

在 `transformPage()` 的实现中，基于当前屏幕显示的页面的 `position` （ `position` 由 `transformPage()` 方法的参数给出）决定哪些页面需要被动画转换，这样我们就能创建自己的动画。

`position` 参数表示特定页面相对于屏幕中的页面的位置。它的值在用户滑动页面过程中动态变化。当某一页面填充屏幕，它的值为0。当页面刚向屏幕右侧方向被拖走，它的值为1。如果用户在页面1和页面2间滑动到一半，那么页面1的position为-0.5并且页面2的position为0.5。根据屏幕上页面的position，我们可以通过 `setAlpha()` ，`setTranslationX()` 或 `setScaleY()` 这些方法设定页面属性来自定义滑动动画。

当我们实现了 `PageTransformer` 后，用我们的实现调用 `setPageTransformer()` 来应用这些自定义动画。例如，如果我们有一个叫做 `ZoomOutPageTransformer` 的 `PageTransformer` ，可以这样设置自定义动画：

```
ViewPager mPager = (ViewPager) findViewById(R.id.pager);
...
mPager.setPageTransformer(true, new ZoomOutPageTransformer());
```

详情查看Zoom-out Page Transformer和Depth Page Transformer部分的 `PageTransformer` 视频和例子。

Zoom-out Page Transformer

当在相邻界面滑动时，这个Page Transformer使页面收缩并褪色。当页面越靠近中心，它将渐渐还原到正常大小并且图像渐入。



```
public class ZoomOutPageTransformer implements ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.85f;
    private static final float MIN_ALPHA = 0.5f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();
        int pageHeight = view.getHeight();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 1) { // [-1,1]
            // Modify the default slide transition to shrink the page as well
            float scaleFactor = Math.max(MIN_SCALE, 1 - Math.abs(position));
            float vertMargin = pageHeight * (1 - scaleFactor) / 2;
            float horzMargin = pageWidth * (1 - scaleFactor) / 2;
            if (position < 0) {
                view.setTranslationX(horzMargin - vertMargin / 2);
            } else {
                view.setTranslationX(-horzMargin + vertMargin / 2);
            }

            // Scale the page down (between MIN_SCALE and 1)
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);

            // Fade the page relative to its size.
            view.setAlpha(MIN_ALPHA +
                    (scaleFactor - MIN_SCALE) /
                    (1 - MIN_SCALE) * (1 - MIN_ALPHA));

        } else { // (1,+Infinity]
            // This page is way off-screen to the right.
            view.setAlpha(0);
        }
    }
}
```

Depth Page Transformer

这个Page Transformer使用默认动画的屏幕左滑动画。但是为右滑使用一种“潜藏”效果的动画。潜藏动画将page淡出，并且线性缩小它。



> 注意：在潜藏过程中，默认动画（屏幕滑动）是仍旧发生的，所以你必须用负的X平移来抵消它。例如：

```
view.setTranslationX(-1 * view.getWidth() * position);
```

下面的例子展示了如何抵消默认滑屏动画：

```
public class DepthPageTransformer implements ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.75f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 0) { // [-1,0]
            // Use the default slide transition when moving to the left page
            view.setAlpha(1);
            view.setTranslationX(0);
            view.setScaleX(1);
            view.setScaleY(1);

        } else if (position <= 1) { // (0,1]
            // Fade the page out.
            view.setAlpha(1 - position);

            // Counteract the default slide transition
            view.setTranslationX(pageWidth * -position);

            // Scale the page down (between MIN_SCALE and 1)
            float scaleFactor = MIN_SCALE
                    + (1 - MIN_SCALE) * (1 - Math.abs(position));
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);

        } else { // (1,+Infinity]
            // This page is way off-screen to the right.
            view.setAlpha(0);
        }
    }
}
```

展示Card翻转动画

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/cardflip.html

这节课展示如何使用自定义Fragment动画实现Card翻转动画。Card翻转动画通过模拟Card翻转的效果实现view内容的切换。

下面是card翻转动画的样子：



如果你想直接查看整个例子，下载并运行App样例然后选择Card翻转例子。查看下列文件中的代码实现：

- `src/CardFlipActivity.java`
- `animator/card_flip_right_in.xml`
- `animator/card_flip_right_out.xml`
- `animator/card_flip_left_in.xml`
- `animator/card_flip_left_out.xml`
- `layout/fragment_card_back.xml`
- `layout/fragment_card_front.xml`

创建Animator

创建Card翻转动画，我们需要两个Animator。一个让正面的card的右侧向左翻转渐出，一个让背面的Card向右翻转渐入。我们还需要两个 Animator让背面的card的右侧向左翻转渐入，一个让向右翻转渐入。

card_flip_left_in.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="-180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_left_out.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_right_in.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_right_out.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="-180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

创建View

Card的每一面是一个独立的布局，比如两屏文字，两张图片，或者任何View的组合。然后我们将在应用动画的Fragment里面用到这两个布局。下面的布局创建了展示文本Card的一面：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#a6c"
    android:padding="16dp"
    android:gravity="bottom">

    <TextView android:id="@android:id/text1"
        style="?android:textAppearanceLarge"
        android:textStyle="bold"
        android:textColor="#fff"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_title" />

    <TextView style="?android:textAppearanceSmall"
        android:textAllCaps="true"
        android:textColor="#80ffffff"
        android:textStyle="bold"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_description" />

</LinearLayout>
```

Card另一面显示一个 `ImageView` ：

```
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/image1"
    android:scaleType="centerCrop"
    android:contentDescription="@string/description_image_1" />
```

创建Fragment

为Card正反面创建Fragment，这些类从 `onCreateView()` 方法中分别为每个Framgent返回你之前创建的布局。在想要显示Card的父Activity中，我们可以创建对应的Fragment 实例。下面的例子展示父Activity内嵌套的Fragment：

```java
public class CardFlipActivity extends Activity {
    ...
    /**
     * A fragment representing the front of the card.
     */
    public class CardFrontFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_front, container, false);
        }
    }

    /**
     * A fragment representing the back of the card.
     */
    public class CardBackFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_back, container, false);
        }
    }
}
```

应用card翻转动画

现在，我们需要在父Activity中展示Fragment。为此，首先创建Activity的布局。下面例子创建了一个可以在运行时添加Fragment的 `FrameLayout` 。

```xml
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

在Activity代码中，把先前创建的布局设置成其ContentVew。当Activity创建时展示一个默认的Fragment是个不错的注意。所以下面的Activity样例表明了如何默认显示卡片正面：

```
public class CardFlipActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activity_card_flip);

        if (savedInstanceState == null) {
            getFragmentManager()
                    .beginTransaction()
                    .add(R.id.container, new CardFrontFragment())
                    .commit();
        }
    }
    ...
}
```

既然现在显示了卡片的正面，我们可以在合适时机用翻转动画显示卡片背面了。创建一个方法来显示背面，它需要做下面这些事情：

- 将Fragment转换设置我们刚做的自定义动画
- 用新Fragment替换当前显示的Fragment，并且应用之前创建的动画到该事件中。
- 添加之前显示的Fragment到Fragment的回退栈（back stack）中，所以当用户按下 *Back* 键时，Card会翻转回来。

```
private void flipCard() {
    if (mShowingBack) {
        getFragmentManager().popBackStack();
        return;
    }

    // Flip to the back.

    mShowingBack = true;

    // Create and commit a new fragment transaction that adds the fragment for the bac
k of
    // the card, uses custom animations, and is part of the fragment manager's back st
ack.

    getFragmentManager()
            .beginTransaction()

            // Replace the default fragment animations with animator resources represe
nting
            // rotations when switching to the back of the card, as well as animator
            // resources representing rotations when flipping back to the front (e.g.
when
            // the system Back button is pressed).
            .setCustomAnimations(
                    R.animator.card_flip_right_in, R.animator.card_flip_right_out,
                    R.animator.card_flip_left_in, R.animator.card_flip_left_out)

            // Replace any fragments currently in the container view with a fragment
            // representing the next page (indicated by the just-incremented currentPa
ge
            // variable).
            .replace(R.id.container, new CardBackFragment())

            // Add this transaction to the back stack, allowing users to press Back
            // to get to the front of the card.
            .addToBackStack(null)

            // Commit the transaction.
            .commit();
}
```

缩放View

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/zoom.html

这节课展示怎样实现点击缩放动画，这对相册很有用，他能为相片从缩略图转换成原图并填充屏幕提供动画。

下面展示了触摸缩放动画的效果，它将缩略图扩大并填充屏幕。



如果你想直接查看整个例子，下载并运行App样例然后选择缩放的例子。查看下列文件中的代码实现：

- `src/TouchHighlightImageButton.java` （简单的helper类，当Image Button被按下它显示蓝色高亮）
- `src/ZoomActivity.java`
- `layout/activity_zoom.xml`

创建View

为想要缩放的内容创建一大一小两个版本布局文件。下面的例子为可点击的缩略图新建了一个 `ImageButton` 和一个 `ImageView` 来展示原图：

```xml
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp">

        <ImageButton
            android:id="@+id/thumb_button_1"
            android:layout_width="100dp"
            android:layout_height="75dp"
            android:layout_marginRight="1dp"
            android:src="@drawable/thumb1"
            android:scaleType="centerCrop"
            android:contentDescription="@string/description_image_1" />

    </LinearLayout>

    <!-- This initially-hidden ImageView will hold the expanded/zoomed version of
         the images above. Without transformations applied, it takes up the entire
         screen. To achieve the "zoom" animation, this view's bounds are animated
         from the bounds of the thumbnail button above, to its final laid-out
         bounds.
         -->

    <ImageView
        android:id="@+id/expanded_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:visibility="invisible"
        android:contentDescription="@string/description_zoom_touch_close" />

</FrameLayout>
```

设置缩放动画

一旦实现了布局，我们需要设置触发缩放动画的事件handler。下面的例子为 `ImageButton` 添加了一个 `View.OnClickListener` ，当用户点击按钮时它执行放大动画。

```
public class ZoomActivity extends FragmentActivity {
    // Hold a reference to the current animator,
    // so that it can be canceled mid-way.
    private Animator mCurrentAnimator;

    // The system "short" animation time duration, in milliseconds. This
    // duration is ideal for subtle animations or animations that occur
    // very frequently.
    private int mShortAnimationDuration;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_zoom);

        // Hook up clicks on the thumbnail views.

        final View thumb1View = findViewById(R.id.thumb_button_1);
        thumb1View.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                zoomImageFromThumb(thumb1View, R.drawable.image1);
            }
        });

        // Retrieve and cache the system's default "short" animation time.
        mShortAnimationDuration = getResources().getInteger(
                android.R.integer.config_shortAnimTime);
    }
    ...
}
```

缩放View

我们现在需要适时应用放大动画了。通常来说，我们需要按边界来从小号View放大到大号View。下面的方法展示了如何实现缩放动画：

1. 把高清图像资源设置到已经被隐藏的“放大版”的 `ImageView` 中。为表简单，下面的例子在UI线程中加载了一张大图。但是我们需要在一个单独的线程中来加载以免阻塞UI线程，然后再回到UI线程中设置。理想状况下，图片不要大过屏幕尺寸。
2. 计算 `ImageView` 开始和结束时的边界。
3. 从起始边到结束边同步地动态改变四个位置和大小属性 `X`，`Y`（`SCALE_X` 和 `SCALE_Y`）。这四个动画被加入到了 `AnimatorSet`，所以我们可以让它们一起开始。

4. 缩小则运行相同的动画，但是是在用户点击屏幕放大时的逆向效果。我们可以在 `ImageView` 中添加一个 `View.OnClickListener` 来实现它。当点击时，`ImageView` 缩回到原来缩略图的大小，然后设置它的visibility为 `GONE` 来隐藏。

```
private void zoomImageFromThumb(final View thumbView, int imageResId) {
    // If there's an animation in progress, cancel it
    // immediately and proceed with this one.
    if (mCurrentAnimator != null) {
        mCurrentAnimator.cancel();
    }

    // Load the high-resolution "zoomed-in" image.
    final ImageView expandedImageView = (ImageView) findViewById(
            R.id.expanded_image);
    expandedImageView.setImageResource(imageResId);

    // Calculate the starting and ending bounds for the zoomed-in image.
    // This step involves lots of math. Yay, math.
    final Rect startBounds = new Rect();
    final Rect finalBounds = new Rect();
    final Point globalOffset = new Point();

    // The start bounds are the global visible rectangle of the thumbnail,
    // and the final bounds are the global visible rectangle of the container
    // view. Also set the container view's offset as the origin for the
    // bounds, since that's the origin for the positioning animation
    // properties (X, Y).
    thumbView.getGlobalVisibleRect(startBounds);
    findViewById(R.id.container)
            .getGlobalVisibleRect(finalBounds, globalOffset);
    startBounds.offset(-globalOffset.x, -globalOffset.y);
    finalBounds.offset(-globalOffset.x, -globalOffset.y);

    // Adjust the start bounds to be the same aspect ratio as the final
    // bounds using the "center crop" technique. This prevents undesirable
    // stretching during the animation. Also calculate the start scaling
    // factor (the end scaling factor is always 1.0).
    float startScale;
    if ((float) finalBounds.width() / finalBounds.height()
            > (float) startBounds.width() / startBounds.height()) {
        // Extend start bounds horizontally
        startScale = (float) startBounds.height() / finalBounds.height();
        float startWidth = startScale * finalBounds.width();
        float deltaWidth = (startWidth - startBounds.width()) / 2;
        startBounds.left -= deltaWidth;
        startBounds.right += deltaWidth;
    } else {
        // Extend start bounds vertically
        startScale = (float) startBounds.width() / finalBounds.width();
        float startHeight = startScale * finalBounds.height();
        float deltaHeight = (startHeight - startBounds.height()) / 2;
        startBounds.top -= deltaHeight;
```

```
        startBounds.bottom += deltaHeight;
    }

    // Hide the thumbnail and show the zoomed-in view. When the animation
    // begins, it will position the zoomed-in view in the place of the
    // thumbnail.
    thumbView.setAlpha(0f);
    expandedImageView.setVisibility(View.VISIBLE);

    // Set the pivot point for SCALE_X and SCALE_Y transformations
    // to the top-left corner of the zoomed-in view (the default
    // is the center of the view).
    expandedImageView.setPivotX(0f);
    expandedImageView.setPivotY(0f);

    // Construct and run the parallel animation of the four translation and
    // scale properties (X, Y, SCALE_X, and SCALE_Y).
    AnimatorSet set = new AnimatorSet();
    set
            .play(ObjectAnimator.ofFloat(expandedImageView, View.X,
                    startBounds.left, finalBounds.left))
            .with(ObjectAnimator.ofFloat(expandedImageView, View.Y,
                    startBounds.top, finalBounds.top))
            .with(ObjectAnimator.ofFloat(expandedImageView, View.SCALE_X,
            startScale, 1f)).with(ObjectAnimator.ofFloat(expandedImageView,
                    View.SCALE_Y, startScale, 1f));
    set.setDuration(mShortAnimationDuration);
    set.setInterpolator(new DecelerateInterpolator());
    set.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationEnd(Animator animation) {
            mCurrentAnimator = null;
        }

        @Override
        public void onAnimationCancel(Animator animation) {
            mCurrentAnimator = null;
        }
    });
    set.start();
    mCurrentAnimator = set;

    // Upon clicking the zoomed-in image, it should zoom back down
    // to the original bounds and show the thumbnail instead of
    // the expanded image.
    final float startScaleFinal = startScale;
    expandedImageView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if (mCurrentAnimator != null) {
                mCurrentAnimator.cancel();
            }
```

```
            // Animate the four positioning/sizing properties in parallel,
            // back to their original values.
            AnimatorSet set = new AnimatorSet();
            set.play(ObjectAnimator
                        .ofFloat(expandedImageView, View.X, startBounds.left))
                        .with(ObjectAnimator
                                .ofFloat(expandedImageView,
                                        View.Y,startBounds.top))
                        .with(ObjectAnimator
                                .ofFloat(expandedImageView,
                                        View.SCALE_X, startScaleFinal))
                        .with(ObjectAnimator
                                .ofFloat(expandedImageView,
                                        View.SCALE_Y, startScaleFinal));
            set.setDuration(mShortAnimationDuration);
            set.setInterpolator(new DecelerateInterpolator());
            set.addListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    thumbView.setAlpha(1f);
                    expandedImageView.setVisibility(View.GONE);
                    mCurrentAnimator = null;
                }

                @Override
                public void onAnimationCancel(Animator animation) {
                    thumbView.setAlpha(1f);
                    expandedImageView.setVisibility(View.GONE);
                    mCurrentAnimator = null;
                }
            });
            set.start();
            mCurrentAnimator = set;
        }
    });
}
```

布局变更动画

> 编写:XizhiXu - 原文:http://developer.android.com/training/animation/layout.html

布局动画是一种预加载动画，系统在每次改变布局配置时运行它。我们需要做的仅是在布局文件里设置属性告诉Android系统为这些布局的变更应用动画，然后系统的默认动画便会执行。

> 小贴士: 如果你想补充自定义布局动画，创建 `LayoutTransition` 对象，然后用 `setLayoutTransition()` 方法把它加到布局中。

下面的例子在一个list中添加一项的默认布局动画：



如果你想直接查看整个例子，下载 App 样例并运行然后选择布局渐变的例子。查看下列文件中的代码实现：

- `src/LayoutChangesActivity.java`
- `layout/activity_layout_changes.xml`
- `menu/activity_layout_changes.xml`

创建布局

在Activity的XML布局文件中，为想开启动画的布局设置 `android:animateLayoutChanges` 属性为 `true` 。例如：

```
<LinearLayout android:id="@+id/container"
    android:animateLayoutChanges="true"
    ...
/>
```

从布局中添加，更新或删除项目

现在，我们需要做的就是添加，删除或更新布局里的项目，然后这些项目就会自动显示动画：

```
private ViewGroup mContainerView;
...
private void addItem() {
    View newView;
    ...
    mContainerView.addView(newView, 0);
}
```

Android网络连接与云服务

> 编写:kesenhoo - 原文:http://developer.android.com/training/building-connectivity.html

这些课程介绍如何让我们的 app 连接到用户设备之外的世界。我们将会学习如何连接到这个区域的其他设备，连接到互联网，以及备份和同步应用程序数据等等。

无线连接设备 - Connecting Devices Wirelessly

如何使用网络服务发现（Network Service Discovery）找到并连接当地设备，以及如何用Wi-Fi创建点对点连接.

执行网络操作 - Performing Network Operations

如何创建一个网络连接，监视连接的变化，以及使用 XML 数据执行事务.

传输数据时避免消耗大量电量 - Transferring Data Without Draining the Battery

如何在 app 执行下载和其他网络事务时最小化对电池的消耗。

云同步 - Syncing to the Cloud

如何同步和备份应用程序和用户数据到云中的远程web服务，以及如何恢复数据到多个设备。

解决云同步的保存冲突：Resolving Cloud Save Conflicts

如何为 app 设计一个健壮的存储数据到云的冲突解决策略。

使用Sycn Adapter传输数据 - Transferring Data Using Sync Adapters

如何使用 Android sync adapter 框架在云和设备间传输数据。

使用Volley传输网络数据 - Transmitting Network Data Using Volley

如何使用 Volley 通过网络执行快速可扩展的UI操作。

无线连接设备

> 编写:acenodie - 原文:http://developer.android.com/training/connect-devices-wirelessly/index.html

除了能够在云端通信，Android 的无线 API 也允许同一局域网中的设备进行通信，甚至没有连接到网络上，而是物理上隔得很近，也可以相互通信。此外，网络服务发现（Network Service Discovery，简称NSD）可以进一步通过允许应用程序运行能相互通信的服务去寻找附近运行相同服务的设备。把这个功能整合到我们的应用中，可以提供许多功能，如在同一个房间，用户玩游戏，可以利用 NSD 实现从一个网络摄像头获取图像，或远程登录到在同一网络中的其他机器。

本节课介绍了一些使我们的应用程序能够寻找和连接其他设备的主要 API。具体地说，它介绍了用于发现可用服务的 NSD API 和能实现点对点无线连接的无线点对点（the Wi-Fi Peer-to-Peer，简称 Wi-Fi P2P）API。本节课也将告诉我们怎样将 NSD 和 Wi-Fi P2P 结合起来去检测其他设备所提供的服务。当检测到时，连接到相应的设备上。即使设备都没有连接到一个网络中。

Lessons

使用网络服务发现

学习如何广播由我们自己的应用程序提供的服务，如何发现在本地网络上提供的服务，并用 NSD 获取我们将要连接的服务的详细信息。

使用 **WiFi** 建立 **P2P** 连接

学习如何获取附近的对等设备，如何创建一个设备接入点，如何连接到其他具有 Wi-Fi P2P 连接功能的设备。

使用 **WiFi P2P** 发现服务

学习如何使用 WiFi P2P 服务去发现附近的不在同一个网络的服务。

使用网络服务发现

> 编写:naizhengtan - 原文:http://developer.android.com/training/connect-devices-wirelessly/nsd.html

添加网络服务发现（Network Service Discovery）到我们的 app 中，可以使我们的用户辨识在局域网内支持我们的 app 所请求的服务的设备。这种技术在点对点应用中能够提供大量帮助，例如文件共享、联机游戏等。Android 的网络服务发现（NSD）API 大大降低实现上述功能的难度。

本讲将简要介绍如何创建 NSD 应用，使其能够在本地网络内广播自己的名称和连接信息，并且扫描其它正在做同样事情的应用信息。最后，将介绍如何连接运行着同样应用的另一台设备。

注册 NSD 服务

> **Note:** 这一步骤是选做的。如果我们并不关心在本地网络上广播 app 服务，那么我们可以跳过这一步，直接尝试发现网络中的服务。

在局域网内注册自己服务的第一步是创建 NsdServiceInfo 对象。此对象包含的信息能够帮助网络中的其他设备决定是否要连接到我们所提供的服务。

```
public void registerService(int port) {
    // Create the NsdServiceInfo object, and populate it.
    NsdServiceInfo serviceInfo  = new NsdServiceInfo();

    // The name is subject to change based on conflicts
    // with other services advertised on the same network.
    serviceInfo.setServiceName("NsdChat");
    serviceInfo.setServiceType("_http._tcp.");
    serviceInfo.setPort(port);
    ....
}
```

这段代码将服务命名为“NsdChat”。该名称将对所有局域网络中使用 NSD 查找本地服务的设备可见。需要注意的是，在网络内该名称必须是独一无二的。Android 系统会自动处理冲突的服务名称。如果同时有两个名为“NsdChat”的应用，其中一个会被自动转换为类似“NsdChat(1)”这样的名称。

第二个参数设置了服务类型，即指定应用使用的协议和传输层。语法是“_< protocol >._< transportlayer >”。在上面的代码中，服务使用了TCP协议上的HTTP协议。想要提供打印服务（例如，一台网络打印机）的应用应该将服务的类型设置为“_ipp._tcp”。

> **Note:** 互联网编号分配机构（International Assigned Numbers Authority，简称 IANA）提供用于服务发现协议（例如 NSD 和 Bonjour）的官方服务种类列表。我们可以下载该列表了解相应的服务名称和端口号码。如果我们想起用新的服务种类，应该向 IANA 官方提交申请。

当为我们的服务设置端口号时，应该尽量避免将其硬编码在代码中，以防止与其他应用产生冲突。例如，如果我们的应用仅仅使用端口1337，就可能与其他使用1337端口的应用发生冲突。解决方法是，不要硬编码，使用下一个可用的端口。不必担心其他应用无法知晓服务的端口号，因为该信息将包含在服务的广播包中。接收到广播后，其他应用将从广播包中得知服务端口号，并通过端口连接到我们的服务上。

如果使用的是 socket，那么我们可以将端口设置为 0，来初始化 socket 到任意可用的端口。

```
public void initializeServerSocket() {
    // Initialize a server socket on the next available port.
    mServerSocket = new ServerSocket(0);

    // Store the chosen port.
    mLocalPort =  mServerSocket.getLocalPort();
    ...
}
```

现在，我们已经成功的创建了 NsdServiceInfo 对象，接下来要做的是实现 RegistrationListener 接口。该接口包含了注册在 Android 系统中的回调函数，作用是通知应用程序服务注册和注销的成功或者失败。

```
public void initializeRegistrationListener() {
    mRegistrationListener = new NsdManager.RegistrationListener() {

        @Override
        public void onServiceRegistered(NsdServiceInfo NsdServiceInfo) {
            // Save the service name.  Android may have changed it in order to
            // resolve a conflict, so update the name you initially requested
            // with the name Android actually used.
            mServiceName = NsdServiceInfo.getServiceName();
        }

        @Override
        public void onRegistrationFailed(NsdServiceInfo serviceInfo, int errorCode) {
            // Registration failed!  Put debugging code here to determine why.
        }

        @Override
        public void onServiceUnregistered(NsdServiceInfo arg0) {
            // Service has been unregistered.  This only happens when you call
            // NsdManager.unregisterService() and pass in this listener.
        }

        @Override
        public void onUnregistrationFailed(NsdServiceInfo serviceInfo, int errorCode)
{
            // Unregistration failed.  Put debugging code here to determine why.
        }
    };
}
```

万事俱备只欠东风，调用 registerService() 方法，真正注册服务。

因为该方法是异步的，所以在服务注册之后的操作都需要在 onServiceRegistered() 方法中进行。

```
public void registerService(int port) {
    NsdServiceInfo serviceInfo  = new NsdServiceInfo();
    serviceInfo.setServiceName("NsdChat");
    serviceInfo.setServiceType("_http._tcp.");
    serviceInfo.setPort(port);

    mNsdManager = Context.getSystemService(Context.NSD_SERVICE);

    mNsdManager.registerService(
            serviceInfo, NsdManager.PROTOCOL_DNS_SD, mRegistrationListener);
}
```

发现网络中的服务

网络充斥着我们的生活，从网络打印机到网络摄像头，再到联网井字棋。网络服务发现是能让我们的应用融入这一切功能的关键。我们的应用需要侦听网络内服务的广播，发现可用的服务，过滤无效的信息。

与注册网络服务类似，服务发现需要两步骤：用相应的回调函数设置发现监听器（Discover Listener），以及调用 discoverServices() 这个异步API。

首先，实例化一个实现 NsdManager.DiscoveryListener 接口的匿名类。下列代码是一个简单的范例：

```
public void initializeDiscoveryListener() {

    // Instantiate a new DiscoveryListener
    mDiscoveryListener = new NsdManager.DiscoveryListener() {

        //  Called as soon as service discovery begins.
        @Override
        public void onDiscoveryStarted(String regType) {
            Log.d(TAG, "Service discovery started");
        }

        @Override
        public void onServiceFound(NsdServiceInfo service) {
            // A service was found!  Do something with it.
            Log.d(TAG, "Service discovery success" + service);
            if (!service.getServiceType().equals(SERVICE_TYPE)) {
                // Service type is the string containing the protocol and
                // transport layer for this service.
                Log.d(TAG, "Unknown Service Type: " + service.getServiceType());
            } else if (service.getServiceName().equals(mServiceName)) {
                // The name of the service tells the user what they'd be
                // connecting to. It could be "Bob's Chat App".
                Log.d(TAG, "Same machine: " + mServiceName);
            } else if (service.getServiceName().contains("NsdChat")){
                mNsdManager.resolveService(service, mResolveListener);
            }
        }

        @Override
        public void onServiceLost(NsdServiceInfo service) {
            // When the network service is no longer available.
            // Internal bookkeeping code goes here.
            Log.e(TAG, "service lost" + service);
        }

        @Override
        public void onDiscoveryStopped(String serviceType) {
            Log.i(TAG, "Discovery stopped: " + serviceType);
        }

        @Override
```

```
        public void onStartDiscoveryFailed(String serviceType, int errorCode) {
            Log.e(TAG, "Discovery failed: Error code:" + errorCode);
            mNsdManager.stopServiceDiscovery(this);
        }

        @Override
        public void onStopDiscoveryFailed(String serviceType, int errorCode) {
            Log.e(TAG, "Discovery failed: Error code:" + errorCode);
            mNsdManager.stopServiceDiscovery(this);
        }
    };
}
```

NSD API 通过使用该接口中的方法通知用户程序发现何时开始、何时失败以及何时找到可用服务和何时服务丢失（丢失意味着“不再可用”）。在上述代码中，当发现了可用的服务时，程序做了几次检查。

1. 比较找到服务的名称与本地服务的名称，判断设备是否获得自己的（合法的）广播。
2. 检查服务的类型，确认这个类型我们的应用是否可以接入。
3. 检查服务的名称，确认是否接入了正确的应用。

我们并不需要每次都检查服务名称，仅当我们想要接入特定的应用时需要检查。例如，应用只想与运行在其他设备上的相同应用通信。然而，如果应用仅仅想接入到一台网络打印机，那么看到服务类型是“_ipp._tcp”的服务就足够了。

当配置好监听器后，调用 discoverService() 函数，其参数包括试图发现的服务种类、发现使用的协议、以及上一步创建的监听器。

```
mNsdManager.discoverServices(
        SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD, mDiscoveryListener);
```

连接到网络上的服务

当我们的应用发现了网上可接入的服务，首先需要调用 resolveService() 方法，以确定服务的连接信息。实现 NsdManager.ResolveListener 对象并将其传入 `resolveService()` 方法，并使用这个 `NsdManager.ResolveListener` 对象获得包含连接信息的 NsdSerServiceInfo。

```
public void initializeResolveListener() {
    mResolveListener = new NsdManager.ResolveListener() {

        @Override
        public void onResolveFailed(NsdServiceInfo serviceInfo, int errorCode) {
            // Called when the resolve fails.  Use the error code to debug.
            Log.e(TAG, "Resolve failed" + errorCode);
        }

        @Override
        public void onServiceResolved(NsdServiceInfo serviceInfo) {
            Log.e(TAG, "Resolve Succeeded. " + serviceInfo);

            if (serviceInfo.getServiceName().equals(mServiceName)) {
                Log.d(TAG, "Same IP.");
                return;
            }
            mService = serviceInfo;
            int port = mService.getPort();
            InetAddress host = mService.getHost();
        }
    };
}
```

当服务解析完成后，我们将获得服务的详细资料，包括其 IP 地址和端口号。此时，我们就可以创建自己网络连接与服务进行通讯。

当程序退出时注销服务

在应用的生命周期中正确的开启和关闭 NSD 服务是十分关键的。在程序退出时注销服务可以防止其他程序因为不知道服务退出而反复尝试连接的行为。另外，服务发现是一种开销很大的操作，应该随着父 Activity 的暂停而停止，当用户返回该界面时再开启。因此，开发者应该重写 Activity 的生命周期函数，并添加按照需要开启和停止服务广播和发现的代码。

```
//In your application's Activity

    @Override
    protected void onPause() {
        if (mNsdHelper != null) {
            mNsdHelper.tearDown();
        }
        super.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (mNsdHelper != null) {
            mNsdHelper.registerService(mConnection.getLocalPort());
            mNsdHelper.discoverServices();
        }
    }

    @Override
    protected void onDestroy() {
        mNsdHelper.tearDown();
        mConnection.tearDown();
        super.onDestroy();
    }

    // NsdHelper's tearDown method
        public void tearDown() {
        mNsdManager.unregisterService(mRegistrationListener);
        mNsdManager.stopServiceDiscovery(mDiscoveryListener);
    }
```

使用 WiFi 建立 P2P 连接

> 编写:naizhengtan - 原文:http://developer.android.com/training/connect-devices-wirelessly/wifi-direct.html

Wi-Fi 点对点（P2P）API 允许应用程序在无需连接到网络和热点的情况下连接到附近的设备。（Android Wi-Fi P2P 使用 Wi-Fi Direct™ 验证程序进行编译）。Wi-Fi P2P 技术使得应用程序可以快速发现附近的设备并与之交互。相比于蓝牙技术，Wi-Fi P2P 的优势是具有较大的连接范围。

本节主要内容是使用 Wi-Fi P2P 技术发现并连接到附近的设备。

配置应用权限

使用 Wi-Fi P2P 技术，需要添加 CHANGE_WIFI_STATE，ACCESS_WIFI_STATE 以及 INTERNET 三种权限到应用的 manifest 文件。Wi-Fi P2P 技术虽然不需要访问互联网，但是它会使用标准的 Java socket（需要 INTERNET 权限）。下面是使用 Wi-Fi P2P 技术需要申请的权限。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.nsdchat"
    ...

    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...
```

设置广播接收器（BroadCast Receiver）和 P2P 管理器

使用 Wi-Fi P2P 的时候，需要侦听当某个事件出现时发出的broadcast intent。在应用中，实例化一个 IntentFilter，并将其设置为侦听下列事件：

WIFI_P2P_STATE_CHANGED_ACTION

指示 Wi-Fi P2P 是否开启

WIFI_P2P_PEERS_CHANGED_ACTION

代表对等节点（peer）列表发生了变化

WIFI_P2P_CONNECTION_CHANGED_ACTION

表明Wi-Fi P2P的连接状态发生了改变

WIFI_P2P_THIS_DEVICE_CHANGED_ACTION

指示设备的详细配置发生了变化

```
private final IntentFilter intentFilter = new IntentFilter();
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //  Indicates a change in the Wi-Fi P2P status.
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);

    // Indicates a change in the list of available peers.
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);

    // Indicates the state of Wi-Fi P2P connectivity has changed.
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION);

    // Indicates this device's details have changed.
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);

    ...
}
```

在 onCreate() 方法的最后，需要获得 WifiPpManager 的实例，并调用它的 initialize() 方法。该方法将返回 WifiP2pManager.Channel 对象。 我们的应用将在后面使用该对象连接 Wi-Fi P2P 框架。

```
@Override

Channel mChannel;

public void onCreate(Bundle savedInstanceState) {
    ....
    mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(this, getMainLooper(), null);
}
```

接下来，创建一个新的 BroadcastReceiver 类侦听系统中 Wi-Fi P2P 状态的变化。在 onReceive() 方法中，加入对上述四种不同 P2P 状态变化的处理。

```
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            // Determine if Wifi P2P mode is enabled or not, alert
            // the Activity.
            int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
            if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
                activity.setIsWifiP2pEnabled(true);
            } else {
                activity.setIsWifiP2pEnabled(false);
            }
        } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {

            // The peer list has changed!  We should probably do something about
            // that.

        } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {

            // Connection state changed!  We should probably do something about
            // that.

        } else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action))
{
            DeviceListFragment fragment = (DeviceListFragment) activity.getFragmentMan
ager()
                    .findFragmentById(R.id.frag_list);
            fragment.updateThisDevice((WifiP2pDevice) intent.getParcelableExtra(
                    WifiP2pManager.EXTRA_WIFI_P2P_DEVICE));

        }
    }
```

最后，在主 activity 开启时，加入注册 intent filter 和 broadcast receiver 的代码，并在 activity 暂停或关闭时，注销它们。上述做法最好放在 onResume() 和 onPause() 方法中。

```
    /** register the BroadcastReceiver with the intent values to be matched */
    @Override
    public void onResume() {
        super.onResume();
        receiver = new WiFiDirectBroadcastReceiver(mManager, mChannel, this);
        registerReceiver(receiver, intentFilter);
    }

    @Override
    public void onPause() {
        super.onPause();
        unregisterReceiver(receiver);
    }
```

初始化对等节点发现（Peer Discovery）

调用 discoverPeers() 开始搜寻附近带有 Wi-Fi P2P 的设备。该方法需要以下参数：

- 上节中调用 WifiP2pManager 的 initialize() 函数获得的 WifiP2pManager.Channel 对象
- 一个对 WifiP2pManager.ActionListener 接口的实现，包括了当系统成功和失败发现所调用的方法。

```
mManager.discoverPeers(mChannel, new WifiP2pManager.ActionListener() {

        @Override
        public void onSuccess() {
            // Code for when the discovery initiation is successful goes here.
            // No services have actually been discovered yet, so this method
            // can often be left blank.  Code for peer discovery goes in the
            // onReceive method, detailed below.
        }

        @Override
        public void onFailure(int reasonCode) {
            // Code for when the discovery initiation fails goes here.
            // Alert the user that something went wrong.
        }
});
```

需要注意的是，这仅仅表示对Peer发现（Peer Discovery）完成初始化。discoverPeers() 方法开启了发现过程并且立即返回。系统会通过调用 WifiP2pManager.ActionListener 中的方法通知应用对等节点发现过程初始化是否正确。同时，对等节点发现过程本身仍然继续运行，直到一条连接或者一个 P2P 小组建立。

获取对等节点列表

在完成对等节点发现过程的初始化后，我们需要进一步获取附近的对等节点列表。第一步是实现 WifiP2pManager.PeerListListener 接口。该接口提供了 Wi-Fi P2P 框架发现的对等节点信息。下列代码实现了相应功能：

```
private List peers = new ArrayList();
    ...

    private PeerListListener peerListListener = new PeerListListener() {
        @Override
        public void onPeersAvailable(WifiP2pDeviceList peerList) {

            // Out with the old, in with the new.
            peers.clear();
            peers.addAll(peerList.getDeviceList());

            // If an AdapterView is backed by this data, notify it
            // of the change.  For instance, if you have a ListView of available
            // peers, trigger an update.
            ((WiFiPeerListAdapter) getListAdapter()).notifyDataSetChanged();
            if (peers.size() == 0) {
                Log.d(WiFiDirectActivity.TAG, "No devices found");
                return;
            }
        }
    }
```

接下来，完善 Broadcast Receiver 的 onReceiver() 方法。 当收到 WIFI_P2P_PEERS_CHANGED_ACTION 事件时， 调用 requestPeer() 方法获取对等节点列表。我们需要将 WifiP2pManager.PeerListListener 传递给 receiver。一种方法是在 broadcast receiver 的构造函数中，将对象作为参数传入。

```
public void onReceive(Context context, Intent intent) {
    ...
    else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {

        // Request available peers from the wifi p2p manager. This is an
        // asynchronous call and the calling activity is notified with a
        // callback on PeerListListener.onPeersAvailable()
        if (mManager != null) {
            mManager.requestPeers(mChannel, peerListListener);
        }
        Log.d(WiFiDirectActivity.TAG, "P2P peers changed");
    }...
}
```

现在，一个带有 WIFI_P2P_PEERS_CHANGED_ACTION action 的 intent 将触发应用对 Peer 列表的更新。

连接一个对等节点

为了连接到一个对等节点，我们需要创建一个新的 WifiP2pConfig 对象，并将要连接的设备信息从表示我们想要连接设备的 WifiP2pDevice 拷贝到其中。然后调用 connect() 方法。

```
@Override
public void connect() {
    // Picking the first device found on the network.
    WifiP2pDevice device = peers.get(0);

    WifiP2pConfig config = new WifiP2pConfig();
    config.deviceAddress = device.deviceAddress;
    config.wps.setup = WpsInfo.PBC;

    mManager.connect(mChannel, config, new ActionListener() {

        @Override
        public void onSuccess() {
            // WiFiDirectBroadcastReceiver will notify us. Ignore for now.
        }

        @Override
        public void onFailure(int reason) {
            Toast.makeText(WiFiDirectActivity.this, "Connect failed. Retry.",
                    Toast.LENGTH_SHORT).show();
        }
    });
}
```

在本段代码中的 WifiP2pManager.ActionListener 实现仅能通知我们初始化的成功或失败。想要监听连接状态的变化，需要实现 WifiP2pManager.ConnectionInfoListener 接口。接口中的 onConnectionInfoAvailable() 回调函数会在连接状态发生改变时通知应用程序。当有多个设备同时试图连接到一台设备时（例如多人游戏或者聊天群），这一台设备将被指定为“群主”（group owner）。

```
@Override
public void onConnectionInfoAvailable(final WifiP2pInfo info) {

    // InetAddress from WifiP2pInfo struct.
    InetAddress groupOwnerAddress = info.groupOwnerAddress.getHostAddress());

    // After the group negotiation, we can determine the group owner.
    if (info.groupFormed && info.isGroupOwner) {
        // Do whatever tasks are specific to the group owner.
        // One common case is creating a server thread and accepting
        // incoming connections.
    } else if (info.groupFormed) {
        // The other device acts as the client. In this case,
        // you'll want to create a client thread that connects to the group
        // owner.
    }
}
```

此时，回头继续完善 broadcast receiver 的 `onReceive()` 方法，并修改对 WIFI_P2P_CONNECTION_CHANGED_ACTION intent 的监听部分的代码。当接收到该 intent 时，调用 requestConnectionInfo() 方法。此方法为异步，所以结果将会被我们提供的 WifiP2pManager.ConnectionInfoListener 所获取。

```
        ...
        } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {

            if (mManager == null) {
                return;
            }

            NetworkInfo networkInfo = (NetworkInfo) intent
                    .getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);

            if (networkInfo.isConnected()) {

                // We are connected with the other device, request connection
                // info to find group owner IP

                mManager.requestConnectionInfo(mChannel, connectionListener);
            }
            ...
```

使用 WiFi P2P 服务发现

> 编写:naizhengtan - 原文:http://developer.android.com/training/connect-devices-wirelessly/nsd-wifi-direct.html

在本章第一节"使用网络服务发现"中介绍了如何在局域网中发现已连接到网络的服务。然而，即使在不接入网络的情况下，Wi-Fi P2P 服务发现也可以使我们的应用直接发现附近的设备。我们也可以向外公布自己设备上的服务。这些能力可以在没有局域网或者网络热点的情况下，在应用间进行通信。

虽然本节所述的 API 与第一节 NSD（Network Service Discovery）的 API 相似，但是具体的实现代码却截然不同。本节将讲述如何通 过Wi-Fi P2P 技术发现其它设备中可用的服务。本节假设读者已经对 Wi-Fi P2P 的 API 有一定了解。

配置 Manifest

使用 Wi-Fi P2P 技术，需要添加 CHANGE_WIFI_STATE、ACCESS_WIFI_STATE 以及 INTERNET 三种权限到应用的 manifest 文件。虽然 Wi-Fi P2P 技术不需要访问互联网，但是它会使用 Java 中的标准 socket，而使用 socket 需要具有 INTERNET 权限，这也是 Wi-Fi P2P 技术需要申请该权限的原因。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.nsdchat"
    ...

    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...
```

添加本地服务

如果我们想提供一个本地服务，就需要在服务发现框架中注册该服务。当本地服务被成功注册，系统将自动回复所有来自附近的服务发现请求。

三步创建本地服务：

1. 新建 WifiP2pServiceInfo 对象
2. 加入相应服务的详细信息
3. 调用 addLocalService() 为服务发现注册本地服务

```
private void startRegistration() {
        //  Create a string map containing information about your service.
        Map record = new HashMap();
        record.put("listenport", String.valueOf(SERVER_PORT));
        record.put("buddyname", "John Doe" + (int) (Math.random() * 1000));
        record.put("available", "visible");

        // Service information.  Pass it an instance name, service type
        // _protocol._transportlayer , and the map containing
        // information other devices will want once they connect to this one.
        WifiP2pDnsSdServiceInfo serviceInfo =
                WifiP2pDnsSdServiceInfo.newInstance("_test", "_presence._tcp", record)
;

        // Add the local service, sending the service info, network channel,
        // and listener that will be used to indicate success or failure of
        // the request.
        mManager.addLocalService(channel, serviceInfo, new ActionListener() {
            @Override
            public void onSuccess() {
                // Command successful! Code isn't necessarily needed here,
                // Unless you want to update the UI or add logging statements.
            }

            @Override
            public void onFailure(int arg0) {
                // Command failed.  Check for P2P_UNSUPPORTED, ERROR, or BUSY
            }
        });
    }
```

发现附近的服务

Android 使用回调函数通知应用程序附近可用的服务，因此首先要做的是设置这些回调函数。新建一个 WifiP2pManager.DnsSdTxtRecordListener 实例监听实时收到的记录（record）。这些记录可以是来自其他设备的广播。当收到记录时，将其中的设备地址和其他相关信息拷贝到当前方法之外的外部数据结构中，供之后使用。下面的例子假设这条记录包含一个带有用户身份的“buddyname”域（field）。

```
final HashMap<String, String> buddies = new HashMap<String, String>();
...
private void discoverService() {
    DnsSdTxtRecordListener txtListener = new DnsSdTxtRecordListener() {
        @Override
        /* Callback includes:
         * fullDomain: full domain name: e.g "printer._ipp._tcp.local."
         * record: TXT record dta as a map of key/value pairs.
         * device: The device running the advertised service.
         */

        public void onDnsSdTxtRecordAvailable(
                String fullDomain, Map record, WifiP2pDevice device) {
                Log.d(TAG, "DnsSdTxtRecord available -" + record.toString());
                buddies.put(device.deviceAddress, record.get("buddyname"));
            }
        };
    ...
}
```

接下来创建 WifiP2pManager.DnsSdServiceResponseListener 对象，用以获取服务的信息。这个对象将接收服务的实际描述以及连接信息。上一段代码构建了一个包含设备地址和“buddyname”键值对的 Map 对象。WifiP2pManager.DnsSdServiceResponseListener 对象使用这些配对信息将 DNS 记录和对应的服务信息对应起来。当上述两个 listener 构建完成后，调用 setDnsSdResponseListeners() 将他们加入到 WifiP2pManager。

```
private void discoverService() {
...

    DnsSdServiceResponseListener servListener = new DnsSdServiceResponseListener() {
        @Override
        public void onDnsSdServiceAvailable(String instanceName, String registrationTy
pe,
                WifiP2pDevice resourceType) {

                // Update the device name with the human-friendly version from
                // the DnsTxtRecord, assuming one arrived.
                resourceType.deviceName = buddies
                        .containsKey(resourceType.deviceAddress) ? buddies
                        .get(resourceType.deviceAddress) : resourceType.deviceName;

                // Add to the custom adapter defined specifically for showing
                // wifi devices.
                WiFiDirectServicesList fragment = (WiFiDirectServicesList) getFragment
Manager()
                        .findFragmentById(R.id.frag_peerlist);
                WiFiDevicesAdapter adapter = ((WiFiDevicesAdapter) fragment
                        .getListAdapter());

                adapter.add(resourceType);
                adapter.notifyDataSetChanged();
                Log.d(TAG, "onBonjourServiceAvailable " + instanceName);
        }
    };

    mManager.setDnsSdResponseListeners(channel, servListener, txtListener);
    ...
}
```

现在调用 addServiceRequest() 创建服务请求。这个方法也需要一个 Listener 报告请求成功与失败。

```
        serviceRequest = WifiP2pDnsSdServiceRequest.newInstance();
        mManager.addServiceRequest(channel,
                serviceRequest,
                new ActionListener() {
                    @Override
                    public void onSuccess() {
                        // Success!
                    }

                    @Override
                    public void onFailure(int code) {
                        // Command failed.  Check for P2P_UNSUPPORTED, ERROR, or BUSY
                    }
                });
```

最后调用 discoverServices()。

```
mManager.discoverServices(channel, new ActionListener() {

    @Override
    public void onSuccess() {
        // Success!
    }

    @Override
    public void onFailure(int code) {
        // Command failed.  Check for P2P_UNSUPPORTED, ERROR, or BUSY
        if (code == WifiP2pManager.P2P_UNSUPPORTED) {
            Log.d(TAG, "P2P isn't supported on this device.");
        else if(...)
            ...
    }
});
```

如果所有部分都配置正确，我们应该就能看到正确的结果了！如果遇到了问题，可以查看 WifiP2pManager.ActionListener 中的回调函数。它们能够指示操作是否成功。我们可以将 debug 的代码放置在 onFailure() 中来诊断问题。其中的一些错误码（Error Code）也许能为我们带来不小启发。下面是一些常见的错误：

P2P_UNSUPPORTED

　　当前的设备不支持 Wi-Fi P2P

BUSY

　　系统忙，无法处理当前请求

ERROR

　　内部错误导致操作失败

执行网络操作

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/network-ops/index.html

这一章会介绍一些基本的网络操作，涉及到网络连接、监视网络连接（包括网络改变）和让用户控制 app 的网络用途。还会介绍如何解析与使用 XML 数据。

这节课包括一个示例应用，展示如何执行常见的网络操作。我们可以下载下面的的范例，并把它作为可重用代码在自己的应用中使用。

NetworkUsage.zip

通过学习这章节的课程，我们将会学习到一些有关于如何创建一个使用最少的网络流量下载并解析数据的高效 app 的基础知识。

你还可以参考下面文章进阶学习:

- Optimizing Battery Life
- Transferring Data Without Draining the Battery
- Web Apps Overview
- Transmitting Network Data Using Volley

> **Node:** 查看使用 Volley 传输网络数据课程获取 Volley 的相关信息，它是一个能帮助 Android apps 更方便快捷地执行网络操作的 HTTP 库。Volly 可以在开源 AOSP 库中找到。Volly 可能会帮助我们简化网络操作，提高我们 app 的网络操作性能。

Lessons

连接到网络 - Connecting to the Network

学习如何连接到网络，选择一个 HTTP client，以及在 UI 线程外执行网络操作。

管理网络的使用情况 - Managing Network Usage

学习如何检查设备的网络连接情况，创建偏好界面来控制网络使用，以及响应连接变化。

解析 XML 数据 - Parsing XML Data

学习如何解析和使用 XML 数据。

连接到网络

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/network-ops/connecting.html

这一课会演示如何实现一个简单的连接到网络的程序。它提供了一些我们在创建即使最简单的网络连接程序时也应该遵循的最佳示例。

请注意，想要执行本课的网络操作首先需要在程序的 manifest 文件中添加以下权限：

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

选择一个 HTTP Client

大多数连接网络的 Android app 会使用 HTTP 来发送与接收数据。Android 提供了两种 HTTP clients：HttpURLConnection 与 Apache HttpClient。二者均支持 HTTPS、流媒体上传和下载、可配置的超时、IPv6 与连接池（connection pooling）。对于 **Android 2.3 Gingerbread** 或更高的版本，推荐使用 **HttpURLConnection**。关于这部分的更多详情，请参考 Android's HTTP Clients。

检查网络连接

在我们的 app 尝试连接网络之前，应通过函数 getActiveNetworkInfo() 和 isConnected() 检测当前网络是否可用。请注意，设备可能不在网络覆盖范围内，或者用户可能关闭 Wi-Fi 与移动网络连接。关于这部分的更多详情，请参考管理网络的使用情况

```
public void myClickHandler(View view) {
    ...
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    } else {
        // display error
    }
    ...
}
```

在一个单独的线程中执行网络操作

网络操作会遇到不可预期的延迟。为了避免造成不好的用户体验，总是在 UI 线程之外单独的线程中执行网络操作。AsyncTask 类提供了一种简单的方式来处理这个问题。这部分的详情，请参考 Multithreading For Performance。

在下面的代码示例中， `myClickHandler()` 方法会执行 `new DownloadWebpageTask().execute(stringUrl)` 。 `DownloadWebpageTask` 是 `AsyncTask` 的子类，它实现了下面两个方法:

- doInBackground() 执行 `downloadUrl()` 方法。它以网页的 URL 作为参数，方法 `downloadUrl()` 获取并处理网页返回的数据。执行完毕后，返回一个结果字符串。
- onPostExecute() 接收结果字符串并把它显示到 UI 上。

```
public class HttpExampleActivity extends Activity {
    private static final String DEBUG_TAG = "HttpExample";
    private EditText urlText;
    private TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        urlText = (EditText) findViewById(R.id.myUrl);
        textView = (TextView) findViewById(R.id.myText);
    }

    // When user clicks button, calls AsyncTask.
    // Before attempting to fetch the URL, makes sure that there is a network connecti
on.
    public void myClickHandler(View view) {
        // Gets the URL from the UI's text field.
        String stringUrl = urlText.getText().toString();
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
        if (networkInfo != null && networkInfo.isConnected()) {
            new DownloadWebpageText().execute(stringUrl);
        } else {
            textView.setText("No network connection available.");
        }
    }

     // Uses AsyncTask to create a task away from the main UI thread. This task takes a

     // URL string and uses it to create an HttpUrlConnection. Once the connection
     // has been established, the AsyncTask downloads the contents of the webpage as
     // an InputStream. Finally, the InputStream is converted into a string, which is
     // displayed in the UI by the AsyncTask's onPostExecute method.
```

```
    private class DownloadWebpageText extends AsyncTask {
       @Override
       protected String doInBackground(String... urls) {

           // params comes from the execute() call: params[0] is the url.
           try {
               return downloadUrl(urls[0]);
           } catch (IOException e) {
               return "Unable to retrieve web page. URL may be invalid.";
           }
       }
       // onPostExecute displays the results of the AsyncTask.
       @Override
       protected void onPostExecute(String result) {
           textView.setText(result);
      }
    }
    ...
}
```

上面这段代码的事件顺序如下:

1. 当用户点击按钮时调用 `myClickHandler()`，app 将指定的 URL 传给 AsyncTask 的子类 `DownloadWebpageTask`。
2. AsyncTask 的 doInBackground() 方法调用 `downloadUrl()` 方法。
3. `downloadUrl()` 方法以一个 URL 字符串作为参数，并用它创建一个 URL 对象。
4. 这个 URL 对象被用来创建一个 HttpURLConnection。
5. 一旦建立连接，HttpURLConnection 对象将获取网页的内容并得到一个 InputStream。
6. InputStream 被传给 `readIt()` 方法，该方法将流转换成字符串。
7. 最后，AsyncTask 的 onPostExecute() 方法将字符串展示在 main activity 的 UI 上。

连接并下载数据

在执行网络交互的线程里面，我们可以使用 HttpURLConnection 来执行一个 GET 类型的操作并下载数据。在调用 `connect()` 之后，我们可以通过调用 `getInputStream()` 来得到一个包含数据的 InputStream 对象。

在下面的代码示例中，doInBackground() 方法会调用 `downloadUrl()`。这个 `downloadUrl()` 方法使用给予的 URL，通过 HttpURLConnection 连接到网络。一旦建立连接后，app 就会使用 `getInputStream()` 来获取包含数据的 InputStream。

```java
// Given a URL, establishes an HttpUrlConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

请注意，`getResponseCode()` 会返回连接的状态码（status code）。这是一种获知额外网络连接信息的有效方式。其中，状态码是 200 则意味着连接成功。

将输入流（InputStream）转换为字符串

InputStream 是一种可读的 byte 数据源。如果我们获得了一个 InputStream，通常会进行解码（decode）或者转换为目标数据类型。例如，如果我们是在下载图片数据，那么可能需要像下面这样解码并展示它：

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

在上面演示的示例中，InputStream 包含的是网页的文本内容。下面会演示如何把 InputStream 转换为字符串，以便显示在 UI 上。

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws IOException, UnsupportedEncod
ingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

管理网络的使用情况

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/network-ops/managing.html

这一课会介绍如何细化管理使用的网络资源。如果我们的程序需要执行大量网络操作，那么应该提供用户设置选项，来允许用户控制程序的数据偏好。例如，同步数据的频率，是否只在连接到 WiFi 才进行下载与上传操作，是否在漫游时使用套餐数据流量等等。这样用户才不大可能在快到达流量上限时，禁止我们的程序获取后台数据，因为他们可以精确控制我们的 app 使用多少数据流量。

关于如何编写一个最小化下载与网络操作对电量影响的程序，请参考：优化电池寿命和高效下载。

示例：NetworkUsage.zip

检查设备的网络连接

设备可以有许多种网络连接。这节课主要关注使用 Wi-Fi 或移动网络连接的情况。关于所有可能的网络连接类型，请看 ConnectivityManager。

通常 Wi-Fi 是比较快的。移动数据通常都是需要按流量计费，会比较贵。通常我们会选择让 app 在连接到 WiFi 时去获取大量的数据。

在执行网络操作之前，检查设备当前连接的网络连接信息是个好习惯。这样可以防止我们的程序在无意间连接使用了非意向的网络频道。如果网络连接不可用，那么我们的应用应该优雅地做出响应。为了检测网络连接，我们需要使用到下面两个类：

- ConnectivityManager：它会回答关于网络连接的查询结果，并在网络连接改变时通知应用程序。
- NetworkInfo：描述一个给定类型（就本节而言是移动网络或 Wi-Fi）的网络接口状态。

这段代码检查了 Wi-Fi 与移动网络的网络连接。它检查了这些网络接口是否可用（也就是说网络是通的）及是否已连接（也就是说网络连接存在，并且可以建立 socket 来传输数据）：

```
private static final String DEBUG_TAG = "NetworkStatusExample";
...
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

请注意我们不应该仅仅靠网络是否可用来做出决策。由于 isConnected() 能够处理片状移动网络（flaky mobile networks），飞行模式和受限制的后台数据等情况，所以我们应该总是在执行网络操作前检查 isConnected()。

一个更简洁的检查网络是否可用的示例如下。getActiveNetworkInfo() 方法返回一个 NetworkInfo 实例，它表示可以找到的第一个已连接的网络接口，如果返回 null，则表示没有已连接的网络接口(意味着网络连接不可用)：

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

我们可以使用 NetworkInfo.DetailedState，来获取更加详细的网络信息，但很少有这样的必要。

管理网络的使用情况

我们可以实现一个偏好设置的 activity ，使用户能直接设置程序对网络资源的使用情况。例如:

- 可以允许用户仅在连接到 Wi-Fi 时上传视频。
- 可以根据诸如网络可用，时间间隔等条件来选择是否做同步的操作。

写一个支持连接网络和管理网络使用的 app，manifest 里需要有正确的权限和 intent filter。

- manifest 文件里包括下面的权限：
    - android.permission.INTERNET——允许应用程序打开网络套接字。
    - android.permission.ACCESS_NETWORK_STATE——允许应用程序访问网络连接信息。

- 我们可以为 ACTION_MANAGE_NETWORK_USAGE action（Android 4.0中引入）声明 intent filter，表示我们的应用定义了一个提供控制数据使用情况选项的 activity。ACTION_MANAGE_NETWORK_USAGE 显示管理指定应用程序网络数据使用情况的设置。当我们的 app 有一个允许用户控制网络使用情况的设置 activity 时，我们应该为 activity 声明这个 intent filter。在章节概览提供的示例应用中，这个 action 被 `SettingsActivity` 类处理，它提供了偏好设置 UI 来让用户决定何时进行下载。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.networkusage"
    ...>

    <uses-sdk android:minSdkVersion="4"
           android:targetSdkVersion="14" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        ...>
        ...
        <activity android:label="SettingsActivity" android:name=".SettingsActivity">
             <intent-filter>
                <action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
                <category android:name="android.intent.category.DEFAULT" />
          </intent-filter>
        </activity>
    </application>
</manifest>
```

实现一个首选项 Activity

正如上面 manifest 片段中看到的那样，`SettingsActivity` 有一个 ACTION_MANAGE_NETWORK_USAGE action 的 intent filter。`SettingsActivity` 是 PreferenceActivity 的子类，它展示一个偏好设置页面（如下两张图）让用户指定以下内容:

- 是否显示每个 XML 提要条目的总结，或者只是每个条目的一个链接。
- 是否在网络连接可用时下载 XML 提要，或者仅仅在 Wi-Fi 下下载。

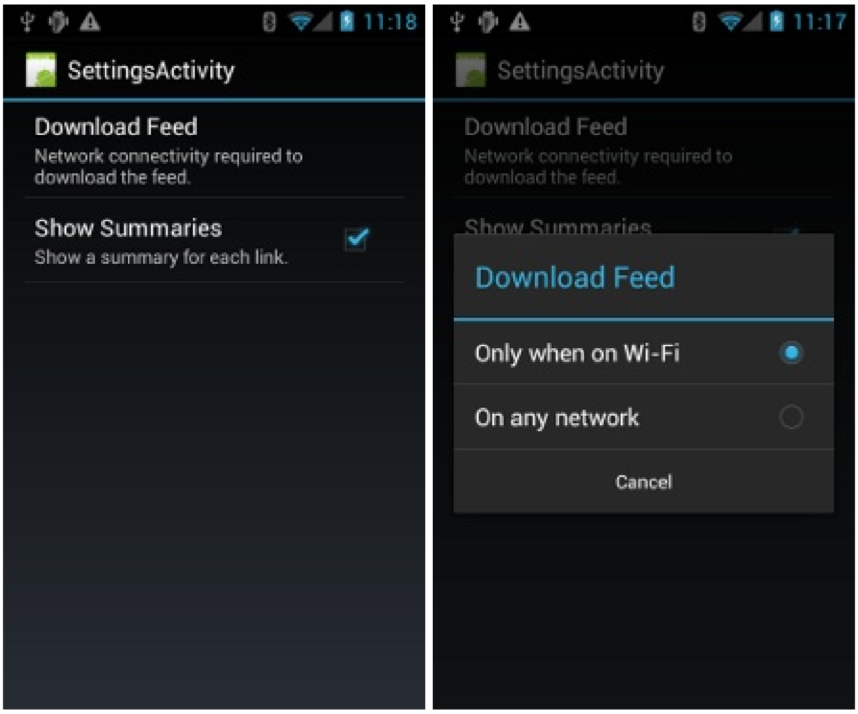


Figure 1. 首选项 activity

下面是 `SettingsActivity`。请注意它实现了 OnSharedPreferenceChangeListener。当用户改变了他的偏好，就会触发 onSharedPreferenceChanged()，这个方法会设置 `refreshDisplay` 为 true（这里的变量存在于自己定义的 activity，见下一部分的代码示例）。这会使得当用户返回到 main activity 的时候进行刷新：

（请注意，代码中的注释，不得不说，Googler 写的 Code 看起来就是舒服）

```
public class SettingsActivity extends PreferenceActivity implements OnSharedPreference
ChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Loads the XML preferences file
        addPreferencesFromResource(R.xml.preferences);
    }

    @Override
    protected void onResume() {
        super.onResume();

        // Registers a listener whenever a key changes
        getPreferenceScreen().getSharedPreferences().registerOnSharedPreferenceChangeL
istener(this);
    }

    @Override
    protected void onPause() {
        super.onPause();

       // Unregisters the listener set in onResume().
       // It's best practice to unregister listeners when your app isn't using them to
 cut down on
       // unnecessary system overhead. You do this in onPause().
       getPreferenceScreen().getSharedPreferences().unregisterOnSharedPreferenceChange
Listener(this);
    }

    // When the user changes the preferences selection,
    // onSharedPreferenceChanged() restarts the main activity as a new
    // task. Sets the the refreshDisplay flag to "true" to indicate that
    // the main activity should update its display.
    // The main activity queries the PreferenceManager to get the latest settings.

    @Override
    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String
key) {
        // Sets refreshDisplay to true so that when the user returns to the main
        // activity, the display refreshes to reflect the new settings.
        NetworkActivity.refreshDisplay = true;
    }
}
```

响应偏好设置的改变

当用户在设置界面改变了偏好，它通常都会对 app 的行为产生影响。在下面的代码示例中，app 会在 `onStart()` 方法中检查偏好设置。如果设置的类型与当前设备的网络连接类型相一致，那么程序就会下载数据并刷新显示。（例如, 如果设置是"Wi-Fi" 并且设备连接了 Wi-Fi）。

（这是一个很好的代码示例，如何选择合适的网络类型进行下载操作）

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagnames=and
roid&sort=newest";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;

    // The user's current network preference setting.
    public static String sPref = null;

    // The BroadcastReceiver that tracks network connectivity changes.
    private NetworkReceiver receiver = new NetworkReceiver();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Registers BroadcastReceiver to track network connection changes.
        IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION
);
        receiver = new NetworkReceiver();
        this.registerReceiver(receiver, filter);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // Unregisters BroadcastReceiver when app is destroyed.
        if (receiver != null) {
            this.unregisterReceiver(receiver);
        }
    }

    // Refreshes the display if the network connection and the
    // pref settings allow it.

    @Override
    public void onStart () {
```

```
        super.onStart();

        // Gets the user's network preference settings
        SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(
this);

        // Retrieves a string value for the preferences. The second parameter
        // is the default value to use if a preference value is not found.
        sPref = sharedPrefs.getString("listPref", "Wi-Fi");

        updateConnectedFlags();

        if(refreshDisplay){
            loadPage();
        }
    }

    // Checks the network connection and sets the wifiConnected and mobileConnected
    // variables accordingly.
    public void updateConnectedFlags() {
        ConnectivityManager connMgr = (ConnectivityManager)
                getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo activeInfo = connMgr.getActiveNetworkInfo();
        if (activeInfo != null && activeInfo.isConnected()) {
            wifiConnected = activeInfo.getType() == ConnectivityManager.TYPE_WIFI;
            mobileConnected = activeInfo.getType() == ConnectivityManager.TYPE_MOBILE;
        } else {
            wifiConnected = false;
            mobileConnected = false;
        }
    }

    // Uses AsyncTask subclass to download the XML feed from stackoverflow.com.
    public void loadPage() {
        if (((sPref.equals(ANY)) && (wifiConnected || mobileConnected))
                || ((sPref.equals(WIFI)) && (wifiConnected))) {
            // AsyncTask subclass
            new DownloadXmlTask().execute(URL);
        } else {
            showErrorPage();
        }
    }
...

}
```

检测网络连接变化

最后一部分是关于 BroadcastReceiver 的子类：`NetworkReceiver`。当设备网络连接改变时，`NetworkReceiver` 会监听到 CONNECTIVITY_ACTION，这时需要判断当前网络连接类型并相应的设置好 `wifiConnected` 与 `mobileConnected`。这样做的结果是下次用户回到 app 时，app 只会下载最新返回的结果。如果 `NetworkActivity.refreshDisplay` 被设置为 `true`，app 会更新显示。

我们需要控制好 BroadcastReceiver 的使用，不必要的声明注册会浪费系统资源。示例应用在 `onCreate()` 中注册 `BroadcastReceiver` `NetworkReceiver`，在 `onDestroy()` 中销毁它。这样做会比在 manifest 里面声明 `<receiver>` 更轻巧。当我们在 manifest 里面声明一个 `<receiver>`，我们的程序可以在任何时候被唤醒，即使我们已经好几个星期没有运行这个程序了。而通过前面的办法注册 `NetworkReceiver`，可以确保用户离开我们的应用之后，应用不会被唤起。如果我们确实要在 manifest 中声明 `<receiver>`，且确保知道何时需要使用到它，那么可以在合适的地方使用 setComponentEnabledSetting() 来开启或者关闭它。

下面是 `NetworkReceiver` 的代码:

```
public class NetworkReceiver extends BroadcastReceiver {

@Override
public void onReceive(Context context, Intent intent) {
    ConnectivityManager conn =  (ConnectivityManager)
        context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = conn.getActiveNetworkInfo();

    // Checks the user prefs and the network connection. Based on the result, decides
whether
    // to refresh the display or keep the current display.
    // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi connecti
on.
    if (WIFI.equals(sPref) && networkInfo != null && networkInfo.getType() == Connecti
vityManager.TYPE_WIFI) {
        // If device has its Wi-Fi connection, sets refreshDisplay
        // to true. This causes the display to be refreshed when the user
        // returns to the app.
        refreshDisplay = true;
        Toast.makeText(context, R.string.wifi_connected, Toast.LENGTH_SHORT).show();

    // If the setting is ANY network and there is a network connection
    // (which by process of elimination would be mobile), sets refreshDisplay to true.
    } else if (ANY.equals(sPref) && networkInfo != null) {
        refreshDisplay = true;

    // Otherwise, the app can't download content--either because there is no network
    // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and there
    // is no Wi-Fi connection.
    // Sets refreshDisplay to false.
    } else {
        refreshDisplay = false;
        Toast.makeText(context, R.string.lost_connection, Toast.LENGTH_SHORT).show();
    }
}
```

解析 XML 数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/basics/network-ops/xml.html

Extensible Markup Language（XML）是一组将文档编码成机器可读形式的规则，也是一种在网络上共享数据的普遍格式。频繁更新内容的网站，比如新闻网站或者博客，经常会提供 XML 提要（XML feed）来使得外部程序可以跟上内容的变化。下载与解析 XML 数据是网络连接相关 app 的一个常见功能。 这一课会介绍如何解析 XML 文档并使用它们的数据。

示例：NetworkUsage.zip

选择一个 Parser

我们推荐 XmlPullParser，它是 Android 上一个高效且可维护的解析 XML 的方法。 Android 上有这个接口的两种实现方式：

- KXmlParser，通过 XmlPullParserFactory.newPullParser() 得到。
- `ExpatPullParser` ，通过 Xml.newPullParser() 得到。

两个选择都是比较好的。下面的示例中是通过 `Xml.newPullParser()` 得到 `ExpatPullParser` 。

分析 Feed

解析一个 feed 的第一步是决定我们需要获取的字段。这样解析器便去抽取出那些需要的字段而忽视其他的字段。

下面的XML片段是章节概览示例 app 中解析的 Feed 的片段。StackOverflow.com 上每一个帖子在 feed 中以包含几个嵌套的子标签的 `entry` 标签的形式出现。

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:creativeCommons="http://backend.userla
nd.com/creativeCommonsRssModule" ...">
<title type="text">newest questions tagged android - Stack Overflow</title>
...
    <entry>
    ...
    </entry>
    <entry>
        <id>http://stackoverflow.com/q/9439999</id>
        <re:rank scheme="http://stackoverflow.com">0</re:rank>
        <title type="text">Where is my data file?</title>
        <category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sort=new
est/tags" term="android"/>
        <category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sort=new
est/tags" term="file"/>
        <author>
            <name>cliff2310</name>
            <uri>http://stackoverflow.com/users/1128925</uri>
        </author>
        <link rel="alternate" href="http://stackoverflow.com/questions/9439999/where-i
s-my-data-file" />
        <published>2012-02-25T00:30:54Z</published>
        <updated>2012-02-25T00:30:54Z</updated>
        <summary type="html">
            <p>I have an Application that requires a data file...</p>

        </summary>
    </entry>
    <entry>
    ...
    </entry>
...
</feed>
```

示例 app 从 `entry` 标签与它的子标签 `title`，`link` 和 `summary` 中提取数据.

实例化 Parser

下一步就是实例化一个 parser 并开始解析的操作。在下面的片段中，一个 parser 被初始化来处理名称空间，并且将 InputStream 作为输入。它通过调用 nextTag() 开始解析，并调用 `readFeed()` 方法，`readFeed()` 方法会提取并处理 app 需要的数据：

```
public class StackOverflowXmlParser {
    // We don't use namespaces
    private static final String ns = null;

    public List parse(InputStream in) throws XmlPullParserException, IOException {
        try {
            XmlPullParser parser = Xml.newPullParser();
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            parser.setInput(in, null);
            parser.nextTag();
            return readFeed(parser);
        } finally {
            in.close();
        }
    }
 ...
}
```

读取Feed

`readFeed()` 方法实际的工作是处理 feed 的内容。它寻找一个 "entry" 的标签作为递归处理整个 feed 的起点。 `readFeed()` 方法会跳过不是 `entry` 的标签。当整个 feed 都被递归处理后， `readFeed()` 会返回一个从 feed 中提取的包含了 `entry` 标签内容（包括里面的数据成员）的 List。然后这个 List 成为 parser 的返回值。

```
private List readFeed(XmlPullParser parser) throws XmlPullParserException, IOException
{
    List entries = new ArrayList();

    parser.require(XmlPullParser.START_TAG, ns, "feed");
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        // Starts by looking for the entry tag
        if (name.equals("entry")) {
            entries.add(readEntry(parser));
        } else {
            skip(parser);
        }
    }
    return entries;
}
```

解析 XML

解析 XML feed 的步骤如下：

1. 正如在上面分析 Feed 所说的，判断出应用中想要的标签。这个例子抽取了 `entry` 标签与它的内部标签 `title`，`link` 和 `summary` 中的数据。
2. 创建下面的方法:
3. 为每一个我们想要获取的标签创建一个 "read" 方法。例如 `readEntry()`，`readTitle()` 等等。解析器从输入流中读取标签。当读取到 `entry`，`title`，`link` 或者 `summary` 标签时，它会为那些标签调用相应的方法。否则，跳过这个标签。

4. 为每一个不同的标签创建提取数据的方法，和使 parser 继续解析下一个标签的方法。例如：

    - 对于 `title` 和 `summary` 标签，解析器调用 `readText()`。这个方法通过调用 `parser.getText()` 来获取数据。

    - 对于 `link` 标签，解析器先判断这个 link 是否是我们想要的类型。然后再使用 `parser.getAttributeValue()` 来获取 link 标签的值。

    - 对于 `entry` 标签，解析器调用 `readEntry()`。这个方法解析 entry 的内部标签并返回一个带有 `title`，`link` 和 `summary` 数据成员的 `Entry` 对象。

5. 一个递归的辅助方法：`skip()`。关于这部分的讨论，请看下面一部分内容：跳过不关心的标签。

下面的代码演示了如何解析 entries，titles，links 与 summaries。

```
public static class Entry {
    public final String title;
    public final String link;
    public final String summary;

    private Entry(String title, String summary, String link) {
        this.title = title;
        this.summary = summary;
        this.link = link;
    }
}

// Parses the contents of an entry. If it encounters a title, summary, or link tag, ha
nds them off
// to their respective "read" methods for processing. Otherwise, skips the tag.
private Entry readEntry(XmlPullParser parser) throws XmlPullParserException, IOExcepti
on {
    parser.require(XmlPullParser.START_TAG, ns, "entry");
    String title = null;
    String summary = null;
```

```
    String link = null;
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        if (name.equals("title")) {
            title = readTitle(parser);
        } else if (name.equals("summary")) {
            summary = readSummary(parser);
        } else if (name.equals("link")) {
            link = readLink(parser);
        } else {
            skip(parser);
        }
    }
    return new Entry(title, summary, link);
}

// Processes title tags in the feed.
private String readTitle(XmlPullParser parser) throws IOException, XmlPullParserExcept
ion {
    parser.require(XmlPullParser.START_TAG, ns, "title");
    String title = readText(parser);
    parser.require(XmlPullParser.END_TAG, ns, "title");
    return title;
}

// Processes link tags in the feed.
private String readLink(XmlPullParser parser) throws IOException, XmlPullParserExcepti
on {
    String link = "";
    parser.require(XmlPullParser.START_TAG, ns, "link");
    String tag = parser.getName();
    String relType = parser.getAttributeValue(null, "rel");
    if (tag.equals("link")) {
        if (relType.equals("alternate")){
            link = parser.getAttributeValue(null, "href");
            parser.nextTag();
        }
    }
    parser.require(XmlPullParser.END_TAG, ns, "link");
    return link;
}

// Processes summary tags in the feed.
private String readSummary(XmlPullParser parser) throws IOException, XmlPullParserExce
ption {
    parser.require(XmlPullParser.START_TAG, ns, "summary");
    String summary = readText(parser);
    parser.require(XmlPullParser.END_TAG, ns, "summary");
    return summary;
}
```

```
// For the tags title and summary, extracts their text values.
private String readText(XmlPullParser parser) throws IOException, XmlPullParserExcepti
on {
    String result = "";
    if (parser.next() == XmlPullParser.TEXT) {
        result = parser.getText();
        parser.nextTag();
    }
    return result;
}
  ...
}
```

跳过不关心的标签

上面描述的 XML 解析步骤中有一步就是跳过不关心的标签，下面演示解析器的 `skip()` 方法:

```
private void skip(XmlPullParser parser) throws XmlPullParserException, IOException {
    if (parser.getEventType() != XmlPullParser.START_TAG) {
        throw new IllegalStateException();
    }
    int depth = 1;
    while (depth != 0) {
        switch (parser.next()) {
        case XmlPullParser.END_TAG:
            depth--;
            break;
        case XmlPullParser.START_TAG:
            depth++;
            break;
        }
    }
}
```

下面解释这个方法如何工作:

- 如果当前事件不是一个 `START_TAG` ，抛出异常。
- 它消耗掉 `START_TAG` 以及接下来的所有内容，包括与开始标签配对的 `END_TAG` 。
- 为了保证方法在遇到正确的 `END_TAG` 时停止，而不是在最开始的 `START_TAG` 后面的第一个标签，方法随时记录嵌套深度。

因此如果目前的标签有子标签, 那么直到解析器已经处理了所有位于 `START_TAG` 与对应的 `END_TAG` 之间的事件之前， `depth` 的值不会为 0。例如，看解析器如何跳过 `<author>` 标签，它有2个子标签， `<name>` 与 `<uri>` ：

- 第一次循环, 在 `<author>` 之后 parser 遇到的第一个标签是 `<name>` 标签的 `START_TAG` 。 `depth` 值变为2。
- 第二次循环, parser 遇到的下一个标签是 `END_TAG` `</name>` 。depth 值变为1。
- 第三次循环, parser 遇到的下一个标签是 `START_TAG` `<uri>` 。depth 值变为2。
- 第四次循环, parser 遇到的下一个标签是 `END_TAG` `</uri>` 。depth 值变为1。
- 第五次同时也是最后一次循环, parser 遇到的下一个标签是 `END_TAG` `</author>` 。depth 值变为0。表明成功跳过了 `<author>` 标签。

使用 XML 数据

示例程序是在 AsyncTask 中获取与解析 XML 数据的。这会在主 UI 线程之外进行处理。当处理完毕后，app 会更新 main activity（ `NetworkActivity` ）的 UI。

在下面示例代码中， `loadPage()` 方法做了下面的事情：

- 初始化一个带有 URL 地址的字符串变量，用来订阅 XML feed。
- 如果用户设置与网络连接都允许，会调用 `new DownloadXmlTask().execute(url)` 。这会初始化一个新的 `DownloadXmlTask` 对象（AsyncTask 的子类）并且开始执行它的 execute() 方法，这个方法会下载并解析 feed，并返回展示在 UI 上的字符串。

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagnames=and
roid&sort=newest";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;
    public static String sPref = null;

    ...

    // Uses AsyncTask to download the XML feed from stackoverflow.com.
    public void loadPage() {

        if((sPref.equals(ANY)) && (wifiConnected || mobileConnected)) {
            new DownloadXmlTask().execute(URL);
        }
        else if ((sPref.equals(WIFI)) && (wifiConnected)) {
            new DownloadXmlTask().execute(URL);
        } else {
            // show error
        }
    }
```

下面展示的是 AsyncTask 的子类，`DownloadXmlTask`，实现了 AsyncTask 的如下方法：

- doInBackground() 执行 `loadXmlFromNetwork()` 方法。它以 feed 的 URL 作为参数。`loadXmlFromNetwork()` 获取并处理 feed。当它完成时，返回一个结果字符串。
- onPostExecute() 接收返回的字符串并将其展示在UI上。

```
// Implementation of AsyncTask used to download XML feed from stackoverflow.com.
private class DownloadXmlTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... urls) {
        try {
            return loadXmlFromNetwork(urls[0]);
        } catch (IOException e) {
            return getResources().getString(R.string.connection_error);
        } catch (XmlPullParserException e) {
            return getResources().getString(R.string.xml_error);
        }
    }

    @Override
    protected void onPostExecute(String result) {
        setContentView(R.layout.main);
        // Displays the HTML string in the UI via a WebView
        WebView myWebView = (WebView) findViewById(R.id.webview);
        myWebView.loadData(result, "text/html", null);
    }
}
```

下面是 `DownloadXmlTask` 中调用的 `loadXmlFromNetwork()` 方法做的事情：

1. 实例化一个 `StackOverflowXmlParser` 。它同样创建一个 `Entry` 对象（ `entries` ）的 List，和 `title` ， `url` ， `summary` ，来保存从 XML feed 中提取的值。
2. 调用 `downloadUrl()` ，它会获取 feed, 并将其作为 InputStream 返回。
3. 使用 `StackOverflowXmlParser` 解析 InputStream。 `StackOverflowXmlParser` 用从 feed 中获取的数据填充 `entries` 的 List。
4. 处理 `entries` 的 List，并将 feed 数据与 HTML 标记结合起来。
5. 返回一个 HTML 字符串，AsyncTask 的 onPostExecute() 方法会将其展示在 main activity 的 UI 上。

```
// Uploads XML from stackoverflow.com, parses it, and combines it with
// HTML markup. Returns HTML string.【这里可以看出应该是Download】
private String loadXmlFromNetwork(String urlString) throws XmlPullParserException, IOE
xception {
    InputStream stream = null;
    // Instantiate the parser
    StackOverflowXmlParser stackOverflowXmlParser = new StackOverflowXmlParser();
    List<Entry> entries = null;
    String title = null;
    String url = null;
    String summary = null;
    Calendar rightNow = Calendar.getInstance();
    DateFormat formatter = new SimpleDateFormat("MMM dd h:mmaa");

    // Checks whether the user set the preference to include summary text
    SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this
```

```
);
    boolean pref = sharedPrefs.getBoolean("summaryPref", false);

    StringBuilder htmlString = new StringBuilder();
    htmlString.append("<h3>" + getResources().getString(R.string.page_title) + "</h3>"
);
    htmlString.append("<em>" + getResources().getString(R.string.updated) + " " +
            formatter.format(rightNow.getTime()) + "</em>");

    try {
        stream = downloadUrl(urlString);
        entries = stackOverflowXmlParser.parse(stream);
    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (stream != null) {
            stream.close();
        }
     }

    // StackOverflowXmlParser returns a List (called "entries") of Entry objects.
    // Each Entry object represents a single post in the XML feed.
    // This section processes the entries list to combine each entry with HTML markup.
    // Each entry is displayed in the UI as a link that optionally includes
    // a text summary.
    for (Entry entry : entries) {
        htmlString.append("<p><a href='");
        htmlString.append(entry.link);
        htmlString.append("'>" + entry.title + "</a></p>");
        // If the user set the preference to include summary text,
        // adds it to the display.
        if (pref) {
            htmlString.append(entry.summary);
        }
    }
    return htmlString.toString();
}

// Given a string representation of a URL, sets up a connection and gets
// an input stream.
【关于Timeout具体应该设置多少，可以借鉴这里的数据，当然前提是一般情况下】
// Given a string representation of a URL, sets up a connection and gets
// an input stream.
private InputStream downloadUrl(String urlString) throws IOException {
    URL url = new URL(urlString);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000 /* milliseconds */);
    conn.setConnectTimeout(15000 /* milliseconds */);
    conn.setRequestMethod("GET");
    conn.setDoInput(true);
    // Starts the query
    conn.connect();
    return conn.getInputStream();
```

```
}
```

传输数据时避免消耗大量电量

> 编写:kesenhoo - 原文:http://developer.android.com/training/efficient-downloads/index.html

在这一章，我们将学习最小化下载，网络连接，尤其是无线电连接对电量的影响。

下面几节课会演示如何使用像缓存（caching）、轮询（polling）和预取（prefetching）这样的技术来调度与执行下载操作。我们还会学习无线电波的 power-use 属性配置是如何影响我们对于在何时，用什么，以何种方式来传输数据的选择。当然这些选择是为了最小化对电量的影响。

我们同样需要阅读 优化电池使用时间

Lesson

优化下载以高效地访问网络

这节课介绍了无线电波状态机（wireless radio state machine），解释了 app 的连接模型（connectivity model）如何与它交互，以及如何最小化数据连接和使用预取（prefetching）和捆绑（bundling）来最小化数据传输对电池消耗的影响。

最小化定期更新造成的影响

这节课我们将了解如何调整刷新频率以最大程度减轻底层无线电波状态机的后台更新所造成的影响。

重复的下载是冗余的

减少下载的最根本途径是只下载我们需要的内容。这节课介绍了消除冗余下载的一些最佳实践。

- 根据网络连接类型来调整下载模式

  不同连接类型对电池电量的影响并不相同。不仅仅是 Wi-Fi 比无线电波更省电，不同的无线电波技术对电量也有不同的影响。

优化下载以高效地访问网络

> 编写:kesenhoo - 原文:http://developer.android.com/training/efficient-downloads/efficient-network-access.html

使用无线电波（wireless radio）进行传输数据很可能是我们 app 最耗电的来源之一。为了最小化网络连接对电量的消耗，懂得连接模式（connectivity model）会如何影响底层的无线电硬件设备是至关重要的。

这节课介绍了无线电波状态机（wireless radio state machine），并解释了 app 的连接模式是如何与状态机进行交互的。然后会提出建议的方法来最小化我们的数据连接，使用预取（prefetching）与捆绑（bundle）的方式进行数据的传输，这些操作都是为了最小化电量的消耗。

无线电波状态机

一个处于完全工作状态的无线电会大量消耗电量，因此需要学习如何在不同能量状态下进行过渡，当无线电没有工作时，节省电量，当需要时尝试最小化与无线电波供电有关的延迟。

典型的 3G 无线电网络有三种能量状态：

1. **Full power**：当无线连接被激活的时候，允许设备以最大的传输速率进行操作。
2. **Low power**：一种中间状态，对电量的消耗差不多是 Full power 状态下的50%。
3. **Standby**：最小的能量状态，没有被激活或者需求的网络连接。

在低功耗和空闲的状态下，电量消耗会显著减少。这里也会介绍重要的网络请求延迟。从 low power 能量状态返回到 full power 大概需要花费1.5秒，从空闲能量状态返回到 full power 状态需要花费2秒。

为了最小化延迟，状态机使用了一种后滞过渡到更低能量状态的机制。下图是一个典型的 3G 无线电波状态机的图示（AT&T电信的一种制式）。

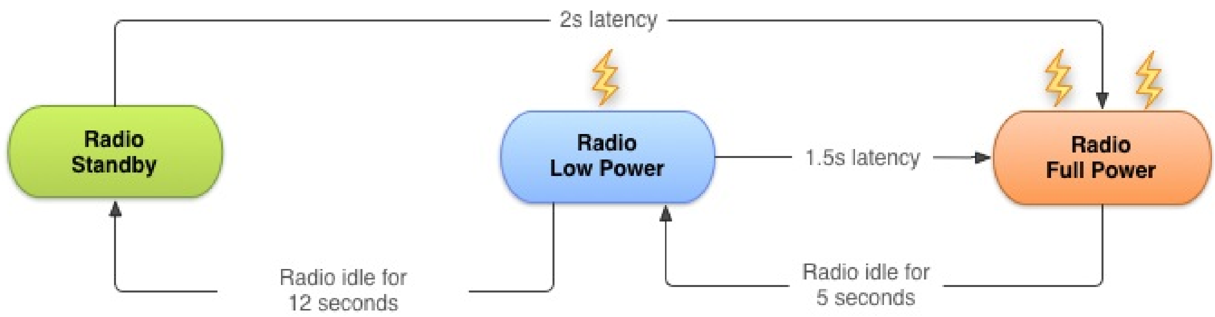


Figure 1. 典型的 3G 无线电状态机

在每一台设备上的无线状态机，特别是相关的传输延迟（“拖尾时间”）和启动延迟，都会根据无线电波的制式（2G、3G、LTE等）不同而改变，并且由设备正在所使用的网络进行定义与配置。

这一课描述了一种典型的 3G 无线电波状态机，数据来源于 AT&T。无论如何，这些原理和最佳实践结果是具有通用性的，在其他的无线电波上同样适用。

这种方法在典型的网页浏览操作上是特别有效的，因为它可以阻止用户在浏览网页时的一些不受欢迎的延迟。相对较短的拖尾时间也保证了当一个网页浏览会话结束的时候，无线电波可以转移到相对较低的能量状态。

不幸的是，这个方法会导致在现代的智能机系统例如 Android 上的 app 效率低下。因为 Android 上的 app 不仅仅可以在前台运行（重点关注延迟），也可以在后台运行（优先处理耗电量）。(无线电波的状态改变会影响到本来的设计，有些想在前台运行的可能会因为切换到低能量状态而影响程序效率。坊间说手机在电量低的状态下无线电波的强度会增大好几倍来保证信号，可能与这个有关。)

App 如何影响无线电波状态机

每次创建一个新的网络连接，无线电波就切换到 full power 状态。在上面典型的 3G 无线电波状态机情况下，无线电波会在传输数据时保持在 full power 的状态（加上一个附加的5秒拖尾时间），再之后会经过12秒的 low power 能量状态。因此对于典型的 3G 设备，每一次数据传输的会话都会导致无线电波消耗大概20秒时间来提取电能。

实际上，这意味着一个每18秒传输1秒非捆绑数据（unbundled data）的 app，会一直保持激活状态（18 = 1秒的传输数据 + 5秒过渡时间回到 low power + 12秒过渡时间回到 standby）。因此，每分钟会消耗18秒 high power 的电量，42秒 low power 的电量。

通过比较，同一个 app，每分钟传输持续3秒的捆绑数据（bundle data），会使得无线电波持续在 high power 状态仅仅8秒，在 low power 状态仅仅12秒钟。

上面第二种传输捆绑数据（bundle data）的例子，可以看到减少了大量的电量消耗。图示如下：

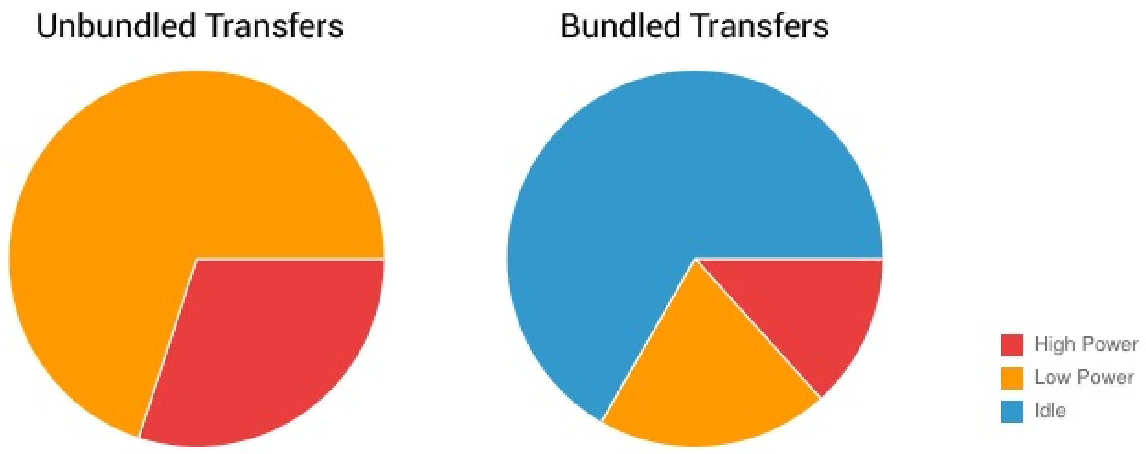


Figure 2. 无线电波使用捆绑数据 vs 无线电波使用非捆绑数据

预取数据

预取数据是一种减少独立数据传输会话数量的有效方法。预取技术指的是在一定时间内，单次连接操作，以最大的下载能力来下载所有用户可能需要的数据。

通过前面的传输数据的技术，减少了大量下载数据所需的无线电波激活时间。这样不仅节省了电量，也改善了延迟，降低了带宽，减少了下载时间。

预取技术通过减少应用里由于在执行一个动作或者查看数据之前等待下载完成造成的延迟，来提高用户体验。

然而，过于频繁地使用预取技术，不仅仅会导致电量消耗快速增长，还有可能预取到一些并不需要的数据，导致增加带宽的使用和下载配额。另外，需要确保预取不会因为 app 等待预取全部完成而延迟应用的启动。从实践的角度，那意味着需要逐步处理数据，或者按照优先级顺序开始进行持续的数据传递，这样会首先下载和处理应用启动时需要的数据。

根据正在下载的数据大小与可能被用到的数据量来决定预取的频率。作一个粗略的估计，根据上面介绍的状态机，对于有50%的机会被当前的用户会话用到的数据，我们可以预取大约6秒(大约1-2Mb)，这大概使得潜在可能要用的数据量与可能已经下载好的数据量相一致。

通常来说，预取1-5Mb会比较好，这种情况下，我们仅仅只需要每隔2-5分钟开始另一段下载。

根据这个原理，大数据的下载，比如视频文件，应该每隔2-5分钟开始另一段下载，这样能有效的预取到下面几分钟内的数据进行预览。

值得注意的是，更进一步的下载应该是是捆绑的（bundled），下一小节将会讲到，批量处理传送和连接，而且上面那些大概的数据与时间可能会根据网络连接的类型与速度有所变化，这将在根据网络连接类型来调整下载模式讲到。

让我们来看一些例子：

一个音乐播放器

我们可以选择预取整个专辑，然而这样在第一首歌曲之后用户会停止听歌，那么就浪费了大量的带宽和电量。

一个比较好的方法是维护正在播放的那首歌曲的缓冲区。对于流媒体音乐，不应该去维护一段连续的数据流，因为这样会使得无线电波一直保持激活状态，而应该考虑用 HTTP 流直播来集中传输音频流，就像上面描述的预取技术一样（下载好2Mb，然后开始一次取出，再去下载下面的2Mb）。

一个新闻阅读器

许多新闻 app 尝试通过只下载新闻标题来减少带宽，完整的文章仅在用户想要读取的时候再去读取，而且文章也会因为太长而刚开始只显示部分信息，等用户下滑时再去读取完整信息。

使用这个方法，无线电波仅仅会在用户点击更多信息的时候才会被激活。但是，在切换文章分类预阅读文章的时候仍然会造成大量潜在的消耗。

一个比较好的方法是在启动的时候预取一个合理数量的数据，比如在启动的时候预取第一条新闻的标题与缩略图信息，确保较短的启动时间。之后继续获取剩余新闻的标题和缩略图信息。同时获取至少在主要标题列表中可用的每篇文章的文本。

另一个方法是预取所有的标题，缩略信息，文章文字，甚至是所有文章的图片——根据既设的后台程序进行逐一获取。这样做的风险是花费了大量的带宽与电量去下载一些不会阅读到的内容，因此应该谨慎使用这种方法。

其中的一个解决方案是，仅当在连接至Wi-Fi或者设备正在充电时，调度到 Full power 状态进行下载。关于这个细节的实现，我们将在后面的根据网络连接类型来调整下载模式课程中介绍。

批量处理传送和连接

每次发起一个连接——不论相关传送数据的大小——当使用典型的 3G 无线网络时，可能会导致无线电波消耗大约20秒的电量。

一个 app 每20秒 ping 一次服务器，仅仅是为了确认 app 正在运行和对用户可见，那么无线电波会无限期地处于开启状态，导致即使在没有实际数据传输的情况下，仍会消耗大量电量。

因此，对传送的数据进行捆绑操作和创建一个等待传输队列就显得非常重要。操作正确的话，可以使得大量的数据集中进行发送，这样使得无线电波的激活时间尽可能的少，同时减少大部分电量的花费。

这样做的潜在好处是尽可能在每次传输数据的会话中尽可能多的传输数据而且减少了会话的次数。

那就意味着我们应该通过队列延迟容忍传送来批量处理我们的传输数据，和抢占调度更新和预取，使得当要求时间敏感传输时，数据会被全部执行。同样地，我们的计划更新和定期的预取应该开启等待传输队列的执行工作。

预取数据部分有一个实际的例子。

以上述使用定期预取的新闻应用为例。新闻阅读器收集分析用户的信息来了解用户的阅读模式，并按照新闻报道的受欢迎程度对新闻进行排序。为了保证新闻最新，应用每个小时会检查更新一次。为了节省带宽，预取缩略图信息和当用户选择某个新闻时下载全部图片，而不去下载每篇文章的所有图片。

在这个例子中，所有在 app 中收集到的分析信息应该捆绑在一起并放入下载队列，而不是一收集到信息就传输。当下载完一张全尺寸的图片或者执行每小时一次更新时，应该传输捆绑好的数据。

任何时间敏感或者按需的传输——例如下载全尺寸图片——应该抢占定期更新。计划好的更新应该与按需传送在同一时间执行。这个方法减小了执行一个定期更新的开销，该定期更新通过下载必要的时间敏感图片的背负式传输实现。

减少连接

通常来说，重用已经存在的网络连接比起重新建立一个新的连接更有效率。重用网络连接同样可以使得在拥挤不堪的网络环境中进行更加智能地作出反应。

当可以捆绑所有请求在一个 GET 里面的时候，不要同时创建多个网络连接或者把多个 GET 请求进行串联。

例如，可以一起请求所有文章的情况下，不要根据多个新闻会话进行多次请求。为传输与服务端和客户端 timeout 相关的终止 / 终止确认数据包，无线电波会保持激活状态，所以如果不需要使用连接时，请立即关闭，而不是等待他们 timeout。

之前说道，如果过早对一个连接执行关闭操作，会导致需要额外的开销来建立一个新的连接。一个有用的妥协是不要立即关闭连接，而是在固定期间的 timeout 之前关闭（即稍微晚点却又不至于到 timeout）。

使用 DDMS Network Traffic Tool 来确定问题的区域

Android DDMS (Dalvik Debug Monitor Server) 包含了一个查看网络使用详情的栏目来允许跟踪 app 的网络请求。使用这个工具，可以监测 app 是在何时，如何传输数据的，从而进行代码的优化。

Figure 3 显示了传输少量数据的网络模型，可以看到每次差不多相隔15秒，这意味着可以通过预取技术或者批量上传来大幅提高效率。

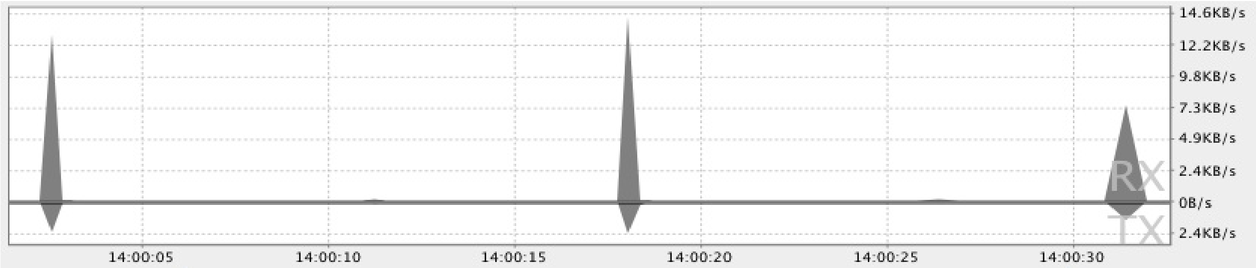


Figure 3. 使用 DDMS 检测网络使用情况

通过监测数据传输的频率与每次传输的数据量，可以查看出哪些位置应该进行优化。通常的，我们会寻找类似短穗状的地方，这些位置可以延迟，或者应该导致一个后来的传输被抢占。

为了更好的检测出问题所在，**Traffic Status API** 允许我们使用 `TrafficStats.setThreadStatsTag()` 方法标记数据传输发生在某个Thread里面，然后可以手动地使用 `tagSocket()` 进行标记或者使用 untagSocket()` 来取消标记，例如：

```
TrafficStats.setThreadStatsTag(0xF00D);
TrafficStats.tagSocket(outputSocket);
// Transfer data using socket
TrafficStats.untagSocket(outputSocket);
```

Apache 的 `HttpClient` 与 `URLConnection` 库可以根据当前的 `getThreadStatusTag()` 值自动给 sockets 加上标记。那些库在通过 keep-alive pools 循环的时候也会为 sockets 加上或者取消标签。

```
TrafficStats.setThreadStatsTag(0xF00D);
try {
  // Make network request using HttpClient.execute()
} finally {
  TrafficStats.clearThreadStatsTag();
}
```

给 Socket 加上标签（Socket tagging）是在 Android 4.0 上才被支持的, 但是实际情况是仅仅会在运行Android 4.0.3 或者更高版本的设备上才会显示。

最小化定期更新造成的影响

> 编写:kesenhoo - 原文:http://developer.android.com/training/efficient-downloads/regular_updates.html

最佳的定期更新频率是不确定的，通常由设备状态，网络连接状态，用户行为与用户显式定义的偏好而决定。

Optimizing Battery Life这一章有讨论如何根据主设备状态来修改更新频率，从而达到编写一个低电量消耗的程序。可执行的操作包括当断开网络连接的时候去关闭后台服务，在电量比较低的时候减少更新的频率等。

这一课会介绍更新频率是多少才会使得更新操作对无线电状态机的影响最小。(C2DM与指数退避算法的使用)

使用 Google Cloud Messaging 来轮询

每次 app 去向 server 询问检查是否有更新操作的时候，都会激活无线电，这样造成了不必要的能量消耗（在3G情况下，会差不多消耗20秒的能量）。

Google Cloud Messaging for Android (GCM) 是一个用来从 server 到特定 app 传输数据的轻量级的机制。使用 GCM，server 会在某个 app 需要获取新数据的时候通知 app 有这个消息。

比起轮询方式（app 为了即时拿到最新的数据需要定时去ping server），GCM 这种由事件驱动的模式会在仅仅有数据更新的时候通知 app 去创建网络连接来获取数据（很显然这样减少了 app 的大量操作，当然也减少了很多电量消耗）。

GCM 需要通过使用持续的 TCP/IP 连接来实现操作。当我们可以实现自己的推送服务，最好使用 GCM（这个地方应该不是传统意义上的固定IP，可以理解为某个会话情况下）。很明显，使用 GCM 既减少了网络连接次数，也优化了带宽，还减少了对电量的消耗。

PS：大陆的 **Google** 框架通常被移除掉，这导致 **GCM** 实际上根本没有办法在大陆的 **App** 上使用。

使用不严格的重复通知和指数避退算法来优化轮询

如果需要使用轮询机制，在不影响用户体验的前提下，设置默认的更新频率当然是越低越好（减少耗电量）。

一个简单的方法是给用户显式修改更新频率的选项，允许用户自己来处理如何平衡数据及时性与电量的消耗。

当设置安排好更新操作后，可以使用不确定重复提醒的方式来允许系统把当前这个操作进行定向移动（比如推迟一会）。

```
int alarmType = AlarmManager.ELAPSED_REALTIME;
long interval = AlarmManager.INTERVAL_HOUR;
long start = System.currentTimeMillis() + interval;

alarmManager.setInexactRepeating(alarmType, start, interval, pi);
```

如果几个提醒都安排在某个点同时被触发，那么就可以使得多个操作在同一个无线电状态下操作完。

如果可以，请设置提醒的类型为 `ELAPSED_REALTIME` 或者 `RTC` 而不是 `_WAKEUP` 。通过一直等待知道手机在提醒通知触发之前不再处于 standby 模式，进一步地减少电量的消耗。

我们还可以通过根据最近 app 被使用的频率来有选择性地减少更新的频率，从而降低这些定期通知的影响。

另一个方法是在 app 在上一次更新操作之后还未被使用的情况下，使用指数退避算法 `exponential back-off algorithm` 来减少更新频率。断言一个最小的更新频率和任何时候使用 app 都去重置频率通常都是有用的方法。例如：

```
SharedPreferences sp =
  context.getSharedPreferences(PREFS, Context.MODE_WORLD_READABLE);

boolean appUsed = sp.getBoolean(PREFS_APPUSED, false);
long updateInterval = sp.getLong(PREFS_INTERVAL, DEFAULT_REFRESH_INTERVAL);

if (!appUsed)
  if ((updateInterval *= 2) > MAX_REFRESH_INTERVAL)
    updateInterval = MAX_REFRESH_INTERVAL;

Editor spEdit = sp.edit();
spEdit.putBoolean(PREFS_APPUSED, false);
spEdit.putLong(PREFS_INTERVAL, updateInterval);
spEdit.apply();

rescheduleUpdates(updateInterval);
executeUpdateOrPrefetch();
```

初始化一个网络连接的花费不会因为是否成功下载了数据而改变。对于那些成功完成是很重要的时间敏感的传输，我们可以使用指数退避算法来减少重复尝试的次数，这样能够避免浪费电量。例如：

```
private void retryIn(long interval) {
  boolean success = attemptTransfer();

  if (!success) {
    retryIn(interval*2 < MAX_RETRY_INTERVAL ?
            interval*2 : MAX_RETRY_INTERVAL);
  }
}
```

另外，对于可以容忍失败连接的传输（例如定期更新），我们可以简单地忽略失败的连接和传输尝试。

笔者结语:这一课讲到GCM与指数退避算法等，其实这些细节很值得我们注意，如果能在实际项目中加以应用，很明显程序的质量上升了一个档次！

重复的下载是冗余的

> 编写:kesenhoo - 原文:http://developer.android.com/training/efficient-downloads/redundant_redundant.html

减少下载的最基本方法是仅仅下载那些我们需要的。从数据的角度看，我们可以通过传递类似上次更新时间这样的参数来制定查询数据的条件。

同样，在下载图片的时候，server 那边最好能够减少图片的大小，而不是让我们下载完整大小的图片。

缓存文件到本地

另一个重要的技术是避免下载重复的数据。可以使用缓存机制来处理这个问题。缓存静态的资源，包括按需下载例如完整的图片（只要合理和兴）。这些缓存的资源需要分开存放，使得我们可以定期地清理这些缓存，从而控制缓存数据的大小。

为了保证 app 不会因为缓存而导致显示的是旧数据，请在缓存中获取数据的同时检测其是否过期，当数据过期的时候，会提示进行刷新。

```
long currentTime = System.currentTimeMillis();

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

long expires = conn.getHeaderFieldDate("Expires", currentTime);
long lastModified = conn.getHeaderFieldDate("Last-Modified", currentTime);

setDataExpirationDate(expires);

if (lastModified < lastUpdateTime) {
  // Skip update
} else {
  // Parse update
}
```

使用这种方法，可以有效保证缓存里面一直是最新的数据。

我们可以缓存非敏感数据到非受管的外部缓存目录（目录会是sdcard下面的 `Android/data/data/com.xxx.xxx/cache` ）：

```
Context.getExternalCacheDir();
```

或者，我们可以使用受管/安全的应用缓存。请注意，当系统的可用存储空间较小时，存放在内存中的数据有可能会被清除（类似: `system/data/data/com.xxx.xxx./cache` ）。

```
Context.getCache();
```

缓存在上面两个地方的文件都会在 app 卸载的时候被清除。

Ps：请注意这点:发现很多应用总是随便在 **sdcard** 下面创建一个目录用来存放缓存，可是这些缓存又不会随着程序的卸载而被删除，这其实是不符合规范，程序都被卸载了，为何还要留那么多垃圾文件，而且这些文件有可能会泄漏一些隐私信息。除非你的程序是音乐下载，拍照程序等等，这些确定程序生成的文件是会被用户需要留下的，不然都应该使用上面的那种方式来获取 **Cache** 目录。

使用 HttpURLConnection 响应缓存

在 `Android 4.0` 里面为 `HttpURLConnection` 增加了一个响应缓存（这是一个很好的减少 http 请求次数的机制，Android 官方推荐使用 HttpURLConnection 而不是 Apache 的 DefaultHttpClient，就是因为前者不仅仅有针对 android 做 http 请求的优化，还在4.0上增加了 Reponse Cache，这进一步提高了效率)。我们可以使用反射机制开启 HTTP response cache，看下面的例子：

```
private void enableHttpResponseCache() {
  try {
    long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
    File httpCacheDir = new File(getCacheDir(), "http");
    Class.forName("android.net.http.HttpResponseCache")
         .getMethod("install", File.class, long.class)
         .invoke(null, httpCacheDir, httpCacheSize);
  } catch (Exception httpResponseCacheNotAvailable) {
    Log.d(TAG, "HTTP response cache is unavailable.");
  }
}
```

上面的示例代码在 Android 4.0 以上的设备上会开启 response cache，同时不会影响到之前的程序。

在cache被开启之后，所有cache中的HTTP请求都可以直接在本地存储中进行响应，并不需要开启一个新的网络连接。被cache起来的response可以被server所确保没有过期，这样就减少了下载所需的带宽。

没有被cached的response会为了方便下次请求而被存储在response cache中。

根据网络连接类型来调整下载模式

> 编写:kesenhoo - 原文:http://developer.android.com/training/efficient-downloads/connectivity_patterns.html

所有的网络类型（Wi-Fi、3G、2G等）对电量的消耗并不是一样的。不仅是 Wi-Fi 电波比无线电波的耗电量要少很多，而且不同的无线电波（3G、2G、LTE......）使用的电量也不同。

使用 Wi-Fi

在大多数情况下，Wi-Fi 电波会在使用相对较低电量的情况下提供一个相对较大的带宽。因此，我们需要争取尽量使用 Wi-Fi 来传递数据。

我们可以使用 Broadcast Receiver 来监听网络连接状态的变化。当切换为 Wi-Fi 时，我们可以进行大量的数据传递操作，例如下载，执行定时的更新操作，甚至是在这个时候暂时加大更新频率。这些内容都可以在前面的课程中找到。

使用更大的带宽来更不频繁地下载更多数据

当通过无线电进行连接的时候，更大的带宽通常伴随着更多的电量消耗。这意味着 LTE（一种4G网络制式）会比 3G 制式更耗电，当然比起 2G 更甚。

从 Lesson 1 我们知道了无线电状态机是怎么回事，通常来说相对更宽的带宽网络制式会有更长的状态切换时间（也就是从 full power 过渡到 standby 有更长一段时间的延迟）。

同时，更高的带宽意味着可以更大量的进行预取，下载更多的数据。也许这个说法不是很直观，因为过渡的时间比较长，而过渡时间的长短我们无法控制，也就是过渡时间的电量消耗差不多是固定了。既然这样，我们在每次传输会话中为了减少更新的频率而把无线电激活的时间拉长，这样显的更有效率。也就是尽量一次性把事情做完，而不是断断续续的请求。

例如：如果 LTE 无线电的带宽与电量消耗都是 3G 无线电的2倍，我们应该在每次会话的时候都下载4倍于 3G 的数据量，或者是差不多 10Mb（前面文章有说明 3G 一般每次下载 2Mb）。当然，下载到这么多数据的时候，我们需要好好考虑预取本地存储的效率并且需要经常刷新预取的缓存。

我们可以使用 connectivity manager 来判断当前激活的无线电波，并且根据不同结果来修改预取操作。

```
ConnectivityManager cm =
 (ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);

TelephonyManager tm =
  (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
  case (ConnectivityManager.TYPE_WIFI):
    PrefetchCacheSize = MAX_PREFETCH_CACHE; break;
  case (ConnectivityManager.TYPE_MOBILE): {
    switch (tm.getNetworkType()) {
      case (TelephonyManager.NETWORK_TYPE_LTE |
            TelephonyManager.NETWORK_TYPE_HSPAP):
        PrefetchCacheSize *= 4;
        break;
      case (TelephonyManager.NETWORK_TYPE_EDGE |
            TelephonyManager.NETWORK_TYPE_GPRS):
        PrefetchCacheSize /= 2;
        break;
      default: break;
    }
    break;
  }
  default: break;
}
```

Ps：想要最大化效率与最小化电量的消耗，需要考虑的东西太多了，通常来说，会根据 **app** 的功能需求来选择有所侧重，那么前提就是需要了解到底哪些对效率的影响比较大,这有利于我们做出最优选择。

云同步

> 编写:kesenhoo，jdneo - 原文:http://developer.android.com/training/cloudsync/index.html

通过为网络连接提供强大的 API，Android Framework 可以帮助我们建立丰富的、具有云功能的 App。这些 App 可以同步数据到远程服务器端，确保我们所有的设备都能保持数据同步，并且重要的数据都能够备份在云端。

本章节会介绍几种不同的策略来实现具有云功能的 App。包括：使用我们自己的后端网络应用进行数据云同步，以及使用云对数据进行备份。这样的话，当用户将我们的 app 安装到一台新的设备上时，他们之前的使用数据就可以得到恢复了。

Lessons

- **使用备份API**

  学习如何将 Backup API 集成到应用中。通过 Backup API 可以将用户数据（比如配置信息、笔记、高分记录等）无缝地在多台设备上进行同步更新。

- **使用Google Cloud Messaging（已废弃）**

  学习如何高效的发送多播消息，如何正确地响应接收到的Google Cloud Messaging (GCM) 消息，以及如何使用GCM消息与服务器进行高效同步。

使用备份API

> 编写:kesenhoo - 原文:http://developer.android.com/training/cloudsync/backupapi.html

当用户购买了一台新的设备或者是对当前的设备做了恢复出厂设置的操作，用户会希望在进行初始化设置的时候，Google Play 能够把之前安装过的应用恢复到设备上。默认情况是，用户的这些期望并不会发生，他们之前的设置与数据都会丢失。

对于一些数据量相对较少的情况（通常少于1MB），例如用户偏好设置、笔记、游戏分数或者是其他的一些状态数据，可以使用 Backup API 来提供一个轻量级的解决方案。这节课会介绍如何将 Backup API 集成到我们的应用当中，以及如何利用 Backup API 将数据恢复到新的设备上。

注册 Android Backup Service

这节课中所使用的 Android Backup Service 需要进行注册。我们可以点击这里进行注册。注册成功后，服务器会提供一段类似于下面的代码，我们需要将它添加到应用的 Manifest 文件中:

```
<meta-data android:name="com.google.android.backup.api_key"
android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
```

请注意，每一个备份 Key 都只能在特定的包名下工作。如果我们有不同的应用需要使用这个方法进行备份，那么需要分别为他们进行注册。

配置 Manifest 文件

使用 Android 的备份服务需要将两个额外的内容添加到应用的 Manifest 文件中。首先，声明备份代理的类名，然后添加一段类似上面的代码作为 Application 标签的子标签。假设我们的备份代理叫作 `TheBackupAgent` ，下面的例子展示了如何在 Manifest 文件中添加这些信息:

```
<application android:label="MyApp"
             android:backupAgent="TheBackupAgent">
    ...
    <meta-data android:name="com.google.android.backup.api_key"
    android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
    ...
</application>
```

编写备份代理

创建备份代理最简单的方法是继承 BackupAgentHelper。 创建这个帮助类实际上非常简便。首先创建一个类，其类名和上述 Manifest 文件中声明的类名一致（本例中，它叫做 `TheBackupAgent` ），然后继承 `BackupAgentHelper` ，之后重写 onCreate() 方法。

在 onCreate() 中创建一个 BackupHelper。这些帮助类是专门用来备份某些数据的。目前 Android Framework 包含了两种帮助类：FileBackupHelper 与 SharedPreferencesBackupHelper。在我们创建一个帮助类并且指向需要备份的数据的时候，仅仅需要使用 addHelper() 方法将它们添加到 `BackupAgentHelper` 当中， 之后再增加一个 Key 用来恢复数据。大多数情况下，完整的实现差不多只需要10行左右的代码。

下面是一个对高分数据进行备份的例子：

```
 import android.app.backup.BackupAgentHelper;
 import android.app.backup.FileBackupHelper;


 public class TheBackupAgent extends BackupAgentHelper {
    // The name of the SharedPreferences file
    static final String HIGH_SCORES_FILENAME = "scores";

    // A key to uniquely identify the set of backup data
    static final String FILES_BACKUP_KEY = "myfiles";

    // Allocate a helper and add it to the backup agent
    @Override
    void onCreate() {
        FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME);
        addHelper(FILES_BACKUP_KEY, helper);
    }
}
```

为了使得程序更加灵活，FileBackupHelper 的构造函数可以带有任意数量的文件名。我们只需简单地通过增加一个额外的参数，就能实现同时对最高分文件与游戏进度文件进行备份，如下所述：

```
    @Override
    void onCreate() {
        FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME, PRO
GRESS_FILENAME);
        addHelper(FILES_BACKUP_KEY, helper);
    }
```

备份用户偏好同样比较简单。和创建 FileBackupHelper 一样来创建一个 SharedPreferencesBackupHelper。在这种情况下，不是添加文件名到构造函数当中，而是添加被应用所使用的 Shared Preference Groups 的名称。下面的例子展示的是，如果高分数据是以 Preference 的形式而非文件的形式存储的，备份代理帮助类应该如何设计：

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.SharedPreferencesBackupHelper;

public class TheBackupAgent extends BackupAgentHelper {
    // The names of the SharedPreferences groups that the application maintains.  The
se
    // are the same strings that are passed to getSharedPreferences(String, int).
    static final String PREFS_DISPLAY = "displayprefs";
    static final String PREFS_SCORES = "highscores";

    // An arbitrary string used within the BackupAgentHelper implementation to
    // identify the SharedPreferencesBackupHelper's data.
    static final String MY_PREFS_BACKUP_KEY = "myprefs";

    // Simply allocate a helper and install it
    void onCreate() {
        SharedPreferencesBackupHelper helper =
                new SharedPreferencesBackupHelper(this, PREFS_DISPLAY, PREFS_SCORES);
        addHelper(MY_PREFS_BACKUP_KEY, helper);
    }
}
```

虽然我们可以根据喜好增加任意数量的备份帮助类到备份代理帮助类中，但是请记住每一种类型的备份帮助类只需要一个就够了。一个 FileBackupHelper 可以处理所有我们想要备份的文件，而一个 SharedPreferencesBackupHelper 则能够处理所有我们想要备份的 Shared Preference Groups。

请求备份

为了请求一个备份，仅仅需要创建一个 BackupManager 实例，然后调用它的 dataChanged() 方法即可：

```
import android.app.backup.BackupManager;
...

public void requestBackup() {
  BackupManager bm = new BackupManager(this);
  bm.dataChanged();
}
```

该调用会告知备份管理器即将有数据会被备份到云端。在之后的某个时间点，备份管理器会执行备份代理的 onBackup() 方法。无论任何时候，只要数据发生了改变，我们都可以去调用它，并且不用担心这样会增加网络的负荷。如果我们在备份正式发生之前请求了两次备份，那么最终备份操作仅仅会出现一次。

恢复备份数据

一般而言，我们不应该手动去请求恢复，而是应该让应用安装到设备上的时候自动进行恢复。然而，如果确实有必要手动去触发恢复，只需要调用 requestRestore() 方法就可以了。

使用Google Cloud Messaging（已废弃）

> 编写:jdneo - 原文:http://developer.android.com/training/cloudsync/gcm.html

谷歌云消息（GCM）是一个用来给Android设备发送消息的免费服务，它可以极大地提升用户体验。利用GCM消息，你的应用可以一直保持更新的状态，同时不会使你的设备在服务器端没有可用更新时，唤醒无线电并对服务器发起轮询（这会消耗大量的电量）。同时，GCM可以让你最多一次性将一条消息发送给1,000个人，使得你可以在恰当地时机很轻松地联系大量的用户，同时大量地减轻你的服务器负担。

这节课将包含一些把GCM集成到应用中的最佳实践方法，前提是假定你已经对该服务的基本实现有了一个了解。如果不是这样的话，你可以先阅读一下：GCM demo app tutorial。

高效地发送多播消息

一个GCM最有用的特性之一是单条消息最多可以发送给1,000个接收者。这个功能可以更加简单地将重要消息发送给你的所有用户群体。例如，比方说你有一条消息需要发送给1,000,000个人，而你的服务器每秒能发送500条消息。如果你的每条消息只能发送给一个接收者，那么整个消息发送过程将会耗时1,000,000/500=2,000秒，大约半小时。然而，如果一条消息可以一次性地发送给1,000个人的话，那么耗时将会是(1,000,000/1,000)/500=2秒。这不仅仅体现出了GCM的实用性，同时对于一些实时消息而言，其重要性也是不言而喻的。就比如灾难预警或者体育比分播报，如果延迟了30分钟，消息的价值就大打折扣了。

想要利用这一功能非常简单。如果你使用的是Java语言版本的GCM helper library，只需要向 `send` 或者 `sendNoRetry` 方法提供一个注册ID的List就行了（不要只给单个的注册ID）：

```
// This method name is completely fabricated, but you get the idea.
List regIds = whoShouldISendThisTo(message);

// If you want the SDK to automatically retry a certain number of times, use the
// standard send method.
MulticastResult result = sender.send(message, regIds, 5);

// Otherwise, use sendNoRetry.
MulticastResult result = sender.sendNoRetry(message, regIds);
```

如果想用除了Java之外的语言实现GCM支持，可以构建一个带有下列头部信息的HTTP POST请求：

```
Authorization: key=YOUR_API_KEY
Content-type: application/json
```

之后将你想要使用的参数编码成一个JSON对象，列出所有在 `registration_ids` 这个Key下的注册ID。下面的代码片段是一个例子。除了 `registration_ids` 之外的所有参数都是可选的，在 `data` 内的项目代表了用户定义的载荷数据，而非GCM定义的参数。这个HTTP POST消息将会发送到： `https://android.googleapis.com/gcm/send` ：

```
{ "collapse_key": "score_update",
   "time_to_live": 108,
   "delay_while_idle": true,
   "data": {
       "score": "4 x 8",
       "time": "15:16.2342"
   },
   "registration_ids":["4", "8", "15", "16", "23", "42"]
}
```

关于更多GCM多播消息的格式，可以阅读：Sending Messages。

对可替换的消息执行折叠

GCM经常被用作为一个触发器，它告诉移动应用向服务器发起链接并更新数据。在GCM中，可以（也推荐）在新消息要替代旧消息时，使用可折叠的消息（Collapsible Messages）。我们用体育比赛作为例子，如果你向所有用户发送了一条包含了当前比赛比分的消息，15分钟之后，又发送了一条消息更新比分，那么第一条消息就没有意义了。对于那些还没有收到第一条消息的用户，就没有必要将这两条消息全部接收下来，何况如果要接收两条消息，那么设备不得不进行两次响应（比如对用户发出通知或警告），但实际上两条消息中只有一条是重要的。

当你定义了一个折叠Key，此时如果有多个消息在GCM服务器中，以队列的形式等待发送给同一个用户，那么只有最后的那一条消息会被发出。对于之前所说的体育比分的例子，这样做能让设备免于处理不必要的任务，也不会让设备对用户造成太多打扰。对于其他的一些场景比如与服务器同步数据（检查邮件接收），这样做的话可以减少设备需要执行同步的次数。例如，如果有10封邮件在服务器中等待被接收，并且有10条GCM消息发送到设备提醒它有新的邮件，那么实际上只需要一个GCM就够了，因为设备可以一次性把10封邮件都同步了。

为了使用这一特性，只需要在你要发出的消息中添加一个消息折叠Key。如果你在使用GCM helper library，那么就使用Message类的 `collapseKey(String key)` 方法。

```
Message message = new Message.Builder(regId)
    .collapseKey("game4_scores") // The key for game 4.
    .ttl(600) // Time in seconds to keep message queued if device offline.
    .delayWhileIdle(true) // Wait for device to become active before sending.
    .addPayload("key1", "value1")
    .addPayload("key2", "value2")
    .build();
```

如果你没有使用GCM helper library，那么就直接在你要构建的POST头部中添加一个字段。将 `collapse_key` 作为字段名，并将Key的名称作为该字段的值。

在GCM消息中嵌入数据

通常， GCM消息被用作为一个触发器，或者用来告诉设备，在服务器或者别的地方有一些待更新的数据。然而，一条GCM消息的大小最大可以有4kb，因此，有时候可以在GCM消息中放置一些简单的数据，这样的话设备就不需要再去和服务器发起连接了。在下列条件都满足的情况下，我们可以将数据放置在GCM消息中：

- 数据的总大小在4kb以内。
- 每一条消息都很重要，且需要保留。
- 这些消息不适用于消息折叠的使用情形。

例如，短消息或者回合制网游中玩家的移动数据等都是将数据直接嵌入在GCM消息中的例子。而电子邮件就是反面例子了，因为电子邮件的数据量一般都大于4kb，而且用户一般不需要对每一封新邮件都收到一个GCM提醒的消息。

同时在发送多播消息时，也可以考虑这一方法，这样的话就不会导致大量用户在接收到GCM的更新提醒后，同时向你的服务器发起连接。

这一策略不适用于发送大量的数据，有这么一些原因：

- 为了防止恶意软件发送垃圾消息，GCM有发送频率的限制。
- 无法保证消息按照既定的发送顺序到达。
- 无法保证消息可以在你发送后立即到达。假设设备每一秒都接收一条消息，消息的大小限制在1K，那么传输速率为8kbps，或者说是1990年代的家庭拨号上网的速度。那么如此大量的消息，一定会让你的应用在Google Play上的评分非常尴尬。

如果恰当地使用，直接将数据嵌入到GCM消息中，可以加速你的应用的“感知速度”，因为这样一来它就不必再去服务器获取数据了。

智能地响应GCM消息

你的应用不应该仅仅对收到的GCM消息进行响应就够了，还应该响应地更智能一些。至于如何响应需要结合具体情况而定。

不要太过激进

当提醒用户去更新数据时，很容易不小心从“有用的消息”变成“干扰消息”。如果你的应用使用状态栏通知，那么应该更新现有的通知，而不是创建第二个。如果你通过铃声或者震动的方式提醒用户，一定要设置一个计时器。不要让应用每分钟的提醒频率超过1次，不然的话用户很可能会不堪其扰而卸载你的应用，关机，甚至把手机扔到河里。

用聪明的办法同步数据，别用笨办法

当使用GCM告知设备有数据需要从服务器下载时，记住你有4kb大小的数据可以和消息一起发出，这可以帮助你的应用做出更智能的响应。例如，如果你有一个支持订阅的阅读应用，而你的用户订阅了100个源，那么这就可以帮助你的应用更智能地决定应该去服务器下载什么数据。下面的例子说明了在GCM载荷中可以发送什么样的数据，以及设备可以做出什么样的反应：

- `refresh` - 你的应用被告知向每一个源请求数据。此时你的应用可以向100个不同的服务器发起获取订阅内容的请求，或者如果你在服务器上有一个聚合服务，那么可以只发送一个请求，将100个源的数据进行打包并让设备获取，这样一次性就完成更新。
- `refresh, freshID` - 一种更好的解决方案，你的应用可以有针对性的完成更新。
- `refresh, freshID, timestamp` - 三种方案中最好的，如果正好用户在收到GCM消息之前手动做了更新，那么应用可以利用时间戳和当前的更新时间进行对比，并决定是否有必要执行下一步的行动。

解决云同步的保存冲突

> 编写:jdneo - 原文:http://developer.android.com/training/cloudsave/conflict-res.html

这篇文章将介绍当应用使用Cloud Save service存储数据到云端时，如何设计一个鲁棒性较高的冲突解决策略。云存储服务允许我们为每一个在Google服务上的应用用户，存储他们的应用数据。应用可以通过使用云存储API，从Android设备，iOS设备或者Web应用恢复或更新这些数据。

云存储中的保存和加载过程非常直接：它只是一个数据和byte数组之间序列化转换，并将这些数组存储在云端的过程。然而，当用户有多个设备，并且有两个以上的设备尝试将它们的数据存储在云端时，这一保存的行为可能会引起冲突，因此我们必须决定应该如何处理这类问题。云端数据的结构在很大程度上决定了冲突解决方案的鲁棒性，所以务必小心地设计我们的数据存储结构，使得冲突解决方案的逻辑可以正确地处理每一种情况。

本篇文章从一些有缺陷的解决方案入手，并解释他们为何具有缺陷。之后会呈现一个可以避免冲突的解决方案。用于讨论的例子关注于游戏，但解决问题的核心思想是可以适用于任何将数据存储于云端的应用的。

冲突时获得通知

OnStateLoadedListener方法负责从Google服务器下载应用的状态数据。回调函数OnStateLoadedListener.onStateConflict用来给应用在本地状态和云端存储的状态发生冲突时，提供了一个解决机制：

```
@Override
public void onStateConflict(int stateKey, String resolvedVersion,
    byte[] localData, byte[] serverData) {
    // resolve conflict, then call mAppStateClient.resolveConflict()
 ...
}
```

此时应用必须决定要保留哪一个数据，或者它自己提交一个新的数据来表示合并后的数据状态，解决冲突的逻辑由我们自己来实现。

我们必须要意识到云存储服务是在后台执行同步的。所以我们应该确保应用能够在创建这一数据的Context之外接收回调。特别地，如果Google Play服务应用在后台检测到了一个冲突，该回调函数会在下一次加载数据时被调用，通常来说会是在下一次用户启动该应用时。

因此，我们的云存储代码和冲突解决代码的设计必须是和当前Context无关的：也就是说当我们拿到了两个彼此冲突的数据，我们必须仅通过数据集内获取的数据去解决冲突，而不依赖于任何其它任何外部Context。

处理简单情况

下面列举一些解决冲突的简单例子。对于很多应用而言，用这些策略或者其变体就足够解决大多数问题了：

新的比旧的更有效：在一些情况下，新的数据可以替代旧的数据。例如，如果数据代表了用户所选择角色的衣服颜色，那么最近的新的选择就应该覆盖老的选择。在这种情况下，我们可能会选择在云存储数据中存储时间戳。当处理这些冲突时，选择时间戳最新的数据（记住要选择一个可靠的时钟，并注意对不同时区的处理）。

一个数据好于其他数据：在一些情况下，我们是可以有方法在若干数据集中选取一个最好的。例如，如果数据代表了玩家在赛车比赛中的最佳时间，那么显然，在冲突发生时，我们应该保留成绩最好的那个数据。

进行合并：有可能通过计算两个数据集的合并版本来解决冲突。例如，我们的数据代表了用户解锁关卡的进度，那么我们需要的数据就是两个冲突数据的并集。通过这个方法，用户的关卡解锁进度就不会丢失了。这里的例子使用了这一策略的一个变形。

为更复杂的情况设计一个策略

当我们的游戏允许玩家收集可交换物品时（比如金币或者经验点数），情况会变得更加复杂一些。我们来假想一个游戏，叫做“金币跑酷”，游戏中的角色通过跑步不断地收集金币使自己变的富有。每个收集到的金币都会加入到玩家的储蓄罐中。

下面的章节将展示三种在多个设备间解决冲突的方案：有两个看上去还不错，可惜最终还是不能适用于所有情况，最后一个解决方案可以解决多个设备间的数据冲突。

第一个尝试：只保存总数

首先，这个问题看上去像是说：云存储的数据只要存储金币的数量就行了。但是如果就只有这些数据是可用的，那么解决冲突的方案将会严重受到限制。此时最佳的方案只能是在冲突发生时存储数值最大的数据。

想一下表1中所展现的场景。假设玩家一开始有20枚硬币，然后在设备A上收集了10个，在设备B上收集了15个。然后设备B将数据存储到了云端。当设备A尝试去存储的时候，冲突发生了。“只保存总数”的冲突解决方案会存储35作为这一数据的值（两数之间最大的）。

表1. 值保存最大的数（不佳的策略）

事件	设备**A**的数据	设备**B**的数据	云端的数据	实际的总数
开始阶段	20	20	20	20
玩家在A设备上收集了10个硬币	30	20	20	30
玩家在B设备上收集了15个硬币	30	35	20	45
设备B将数据存储至云端	30	35	35	45
设备A尝试将数据存储至云端，发生冲突	30	35	35	45
设备A通过选择两数中最大的数来解决冲突	35	35	35	45

这一策略显然会失败：玩家的金币数从20变成35，但实际上玩家总共收集了25个硬币（A设备10个，B设备15个），所以有10个硬币丢失了。只在云端存储硬币的总数是不足以实现一个健壮的冲突解决算法的。

第二个尝试：存储总数和变化值

另一个方法是在存储数据中包括一些额外的数据，如：自上次提交后硬币增加的数量（delta）。在这一方法中，存储的数据可以用一个二元组来表示（T, d），其中T是硬币的总数，而d是硬币增加的数量。

通过这样的数据存储结构，我们的冲突检测算法在鲁棒性上会有更大的提升空间。但是这个方法在某些情况下依然会存在问题。

下面是包含delta数值的冲突解决算法过程：

- 本地数据：（T, d）
- 云端数据：（T', d'）
- 解决后的数据：（T'+d, d）

例如，当我们在本地状态（T, d）和云端状态（T', d）之间发生了冲突时，可以将它们合并成（T'+d, d）。这意味着我们从本地拿出delta数据，并将它和云端的数据结合起来，乍一看，这种方法可以很好的计量多个设备所收集的金币。

该方法看上去很可靠，但它在具有移动网络的环境中难以适用：

- 用户可能在设备不在线时存储数据。这些改变会以队列形式等待手机联网后提交。
- 这个方法的同步机制是用最新的变化覆盖掉任何之前的变化。换句话说，第二次写入的变化会提交到云端（当设备联网了以后），而第一次写入的变化就被忽略了。

为了进一步说明，我们考虑一下表2所列的场景。在表2列出的一系列操作发生后，云端的状态将是（130, +5），最终冲突解决后的状态是（140, +10）。这是不正确的，因为从总体上而言，用户一共在A上收集了110枚硬币而在B上收集了120枚硬币。总数应该为250。

表2. “总数+增量”策略的失败案例

事件	设备**A**的数据	设备**B**的数据	云端的数据	实际的总数
开始阶段	(20, x)	(20, x)	(20, x)	20
玩家在A设备上收集了100个硬币	(120, +100)	(20, x)	(20, x)	120
玩家在A设备上又收集了10个硬币	(130, +10)	(20, x)	(20, x)	130
玩家在B设备上收集了115个硬币	(130, +10)	(125, +115)	(20, x)	245
玩家在B设备上又收集了5个硬币	(130, +10)	(130, +5)	(20, x)	250
设备B将数据存储至云端	(130, +10)	(130, +5)	(130, +5)	250
设备A尝试将数据存储至云端，发生冲突	(130, +10)	(130, +5)	(130, +5)	250
设备A通过将本地的增量和云端的总数相加来解决冲突	(140, +10)	(130, +5)	(140, +10)	250

注：*x*代表与该场景无关的数据

我们可能会尝试在每次保存后不重置增量数据来解决此问题，这样的话在每个设备上第二次存储的数据就能够代表用户至今为止收集到的所有硬币。此时，设备A在第二次本地存储完成后，数据将是（130, +110）而不是（130, +10）。然而，这样做的话就会发生如表3所述的情况：

表3. 算法改进后的失败案例

事件	设备**A**的数据	设备**B**的数据	云端的数据	实际的总数
开始阶段	(20, x)	(20, x)	(20, x)	20
玩家在A设备上收集了100个硬币	(120, +100)	(20, x)	(20, x)	120
设备A将状态存储到云端	(120, +100)	(20, x)	(120, +100)	120
玩家在A设备上又收集了10个硬币	(130, +110)	(20, x)	(120, +100)	130
玩家在B设备上收集了1个硬币	(130, +110)	(21, +1)	(120, +100)	131
设备B尝试向云端存储数据，发生冲突	(130, +110)	(21, +1)	(120, +100)	131
设备B通过将本地的增量和云端的总数相加来解决冲突	(130, +110)	(121, +1)	(121, +1)	131
设备A尝试将数据存储至云端，发生冲突	(130, +110)	(121, +1)	(121, +1)	131
设备A通过将本地的增量和云端的总数相加来解决冲突	(231, +110)	(121, +1)	(231, +110)	131

注：*x*代表与该场景无关的数据

现在我们碰到了另一个问题：我们给予了玩家过多的硬币。这个玩家拿到了211枚硬币，但实际上他只收集了111枚。

解决办法：

分析之前的几次尝试，我们发现这些策略存在这样的缺陷：无法知晓哪些硬币已经计数了，哪些硬币没有被计数，尤其是当多个设备连续提交的时候，算法会出现混乱。

该问题的解决办法是将我们在云端的数据存储结构改为字典类型，使用字符串+整形的键值对。每一个键值对都代表了一个包含硬币的“委托人”，而总数就应该是将所有记录的值加起来。这一设计的宗旨是每个设备有它自己的“委托人”，并且只有设备自己可以把硬币放到它的“委托人”当中。

字典的结构是：(A:a, B:b, C:c, ...)，其中a代表了“委托人”A所拥有的硬币，b是“委托人”B所拥有的硬币，以此类推。

这样的话，新的冲突解决策略算法将如下所示：

- 本地数据：(A:a, B:b, C:c, ...)
- 云端数据：(A:a', B:b', C:c', ...)

- 解决后的数据：(A:max(a,a'), B:max(b,b'), C:max(c,c'), ...)

例如，如果本地数据是(A:20, B:4, C:7)并且云端数据是(B:10, C:2, D:14)，那么解决冲突后的数据将会是(A:20, B:10, C:7, D:14)。当然，应用的冲突解决逻辑可以根据具体的需求而有所差异。比如对于有一些应用，我们可能希望挑选最小的值。

为了测试新的算法，将它应用于任何一个之前提到过的场景。你将会发现它都能取得正确地结果。

表4阐述了这一点，它使用了表3中所提到的场景。注意下面所列出的关键点：

在初始状态，玩家有20枚硬币。该数据准确体现在了所有设备和云端中，我们用字典：（X:20）来代表它，其中X我们不用太多关心，初始化的数据是哪儿来对该问题没有影响。

当玩家在设备A上收集了100枚硬币，这一变化会作为一个字典保存到云端。字典的值是100是因为这就是玩家在设备A上收集的硬币数量。在这一过程中，没有要执行数据的计算（设备A仅仅是将玩家所收集的数据汇报给了云端）。

每一个新的硬币提交会打包成一个与设备关联的字典并保存到云端。例如，假设玩家又在设备A上收集了100枚硬币，那么对应字典的值被更新为110。

最终的结果就是，应用知道了玩家在每个设备上收集硬币的总数。这样它就能轻易地计算出实际的总数了。

表4. 键值对策略的成功应用案例

事件	设备**A**的数据	设备**B**的数据	云端的数据	实际的总数
开始阶段	(X:20, x)	(X:20, x)	(X:20, x)	20
玩家在A设备上收集了100个硬币	(X:20, A:100)	(X:20)	(X:20)	120
设备A将状态存储到云端	(X:20, A:100)	(X:20)	(X:20, A:100)	120
玩家在A设备上又收集了10个硬币	(X:20, A:110)	(X:20)	(X:20, A:100)	130
玩家在B设备上收集了1个硬币	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
设备B尝试向云端存储数据，发生冲突	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
设备B解决冲突	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
设备A尝试将数据存储至云端，发生冲突	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
设备A解决冲突	(X:20, A:110, B:1)	(X:20, A:100, B:1)	(X:20, A:110, B:1)，**total 131**	131

清除你的数据

在云端允许存储数据的大小是有限制的，所以在后续的论述中，我们将会关注如何避免创建过大的词典。一开始，看上去每个设备只会有一条词典记录，即使是非常激进的用户也不太会拥有上千种不同的设备（对应上千条字典记录）。然而， 获取设备ID的方法很难，并且我们认为这是一种不好的实践方式，所以我们应该使用一个安装ID，这更容易获取也更可靠。这样的话就意味着，每一次用户在每台设备安装一次就会产生一个ID。假设每个键值对占据32字节，由于一个个人云存储缓存最多可以有128K的大小，因此最多可以存储4096条记录。

在现实场景中，你的数据可能更加复杂。在这种情况下，存储数据的记录条数也会进一步受到限制。具体而言则需要取决于实现，比如可能需要添加时间戳来指明每条记录是何时修改的。当你检测到有一条记录在过去几个礼拜或者几个月的时间内都没有被修改，那么就可以安全地将金币数据转移到另一条记录中并删除老的记录。

使用Sync Adapter传输数据

> 编写:jdneo - 原文:http://developer.android.com/training/sync-adapters/index.html

如果我们的应用允许 Android 设备和网络服务器之间进行数据同步，那么它无疑将变得更加实用，更加吸引用户的注意。例如，将数据传输到服务器可以实现数据的备份，另一方面，从服务器获取数据可以让用户随时随地都能使用我们的应用。有时候，用户可能会觉得在线编辑他们的数据并将其发送到设备上，会是一件很方便的事情；或者他们有时会希望将收集到的数据上传到一个统一的存储区域中。

尽管我们可以设计一套自己的系统来实现应用中的数据传输，但我们也可以考虑一下使用 Android 的同步适配器框架（Android's Sync Adapter Framework）。该框架可以用来帮助管理数据，自动传输数据，以及协调不同应用间的同步问题。当使用这个框架时，我们可以利用它的一些特性，而这些特性可能是我们自己设计的传输方案中所没有的：

插件架构（**Plug-in Architecture**）：

允许我们以可调用组件的形式，将传输代码添加到系统中。

自动执行（**Automated Execution**）：

允许我们基于不同的准则自动地执行数据传输，比如：当数据变更时，或者每隔固定一段时间，亦或者每天，来自动执行一次数据传输。另外，系统会自动把当前无法执行的传输添加到一个队列中，并且在合适的时候运行它们。

自动网络监测（**Automated Network Checking**）：

系统只在有网络连接的时候才会运行数据传输。

提升电池使用效率：

允许我们将所有的数据传输任务统一地进行一次性批量传输，这样的话多个数据传输任务会在同一段时间内运行。我们应用的数据传输任务也会和其它应用的传输任务相结合，并一起传输。这样做可以减少系统连接网络的次数，进而减少电量的使用。

账户管理和授权：

如果我们的应用需要用户登录授权，那么我们可以将账户管理和授权的功能集成到数据传输组件中。

本系列课程将展示如何创建一个 Sync Adapter，如何创建一个绑定了 Sync Adapter 的服务（Service），如何提供其它组件来帮助我们将 Sync Adapter 集成到框架中，以及如何通过不同的方法来运行 Sync Adapter。

> **Note**：Sync Adapter 是异步执行的，它可以定期且有效地传输数据，但在实时性上一般难以满足要求。如果我们想要实时地传输数据，那么应该在 AsyncTask 或 IntentService 中完成这一任务。

Sample Code

BasicSyncAdapter.zip

Lessons

创建 Stub 授权器

学习如何在我们的应用中添加一个 Sync Adapter 框架需要的账户处理组件。这节课将展示如何简单地创建一个 Stub Authenticator 组件。

创建 Stub Content Provider

学习如何在我们的应用中添加一个 Sync Adapter 框架需要的 Content Provider 组件。在这节课中，假设我们的应用实际上不需要使用 Content Provider，所以它将教我们如何添加一个 Stub 组件。如果我们的应用已经有了一个 Content Provider 组件，那么可以跳过这节课。

创建 Sync Adapter

学习如何将我们的数据传输代码封装到组件当中，并让其可以被 Sync Adapter 框架自动执行。

执行 Sync Adapter

学习如何使用 Sync Adapter 框架激活并调度数据传输。

创建 Stub 授权器

> 编写:jdneo - 原文:http://developer.android.com/training/sync-adapters/creating-authenticator.html

Sync Adapter 框架假定我们的 Sync Adapter 在同步数据时，设备存储端关联了一个账户，且服务器端需要进行登录验证。因此，我们需要提供一个叫做授权器（Authenticator）的组件作为 Sync Adapter 的一部分。该组件会集成在 Android 账户及认证框架中，并提供一个标准的接口来处理用户凭据，比如登录信息。

即使我们的应用不使用账户，我们仍然需要提供一个授权器组件。在这种情况下，授权器所处理的信息将被忽略，所以我们可以提供一个包含了方法存根（Stub Method）的授权器组件。同时我们需要提供一个绑定 Service，来允许 Sync Adapter 框架调用授权器的方法。

这节课将展示如何定义一个能够满足 Sync Adapter 框架要求的 Stub 授权器。如果我们想要提供可以处理用户账户的实际的授权器，可以阅读：AbstractAccountAuthenticator。

添加一个 Stub 授权器组件

要在应用中添加一个 Stub 授权器，首先我们需要创建一个继承 AbstractAccountAuthenticator 的类，在所有需要重写的方法中，我们不进行任何处理，仅返回 null 或者抛出异常。

下面的代码片段是一个 Stub 授权器的例子：

```
/*
 * Implement AbstractAccountAuthenticator and stub out all
 * of its methods
 */
public class Authenticator extends AbstractAccountAuthenticator {
    // Simple constructor
    public Authenticator(Context context) {
        super(context);
    }
    // Editing properties is not supported
    @Override
    public Bundle editProperties(
            AccountAuthenticatorResponse r, String s) {
        throw new UnsupportedOperationException();
    }
    // Don't add additional accounts
    @Override
    public Bundle addAccount(
            AccountAuthenticatorResponse r,
```

```
            String s,
            String s2,
            String[] strings,
            Bundle bundle) throws NetworkErrorException {
        return null;
    }
    // Ignore attempts to confirm credentials
    @Override
    public Bundle confirmCredentials(
            AccountAuthenticatorResponse r,
            Account account,
            Bundle bundle) throws NetworkErrorException {
        return null;
    }
    // Getting an authentication token is not supported
    @Override
    public Bundle getAuthToken(
            AccountAuthenticatorResponse r,
            Account account,
            String s,
            Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
    // Getting a label for the auth token is not supported
    @Override
    public String getAuthTokenLabel(String s) {
        throw new UnsupportedOperationException();
    }
    // Updating user credentials is not supported
    @Override
    public Bundle updateCredentials(
            AccountAuthenticatorResponse r,
            Account account,
            String s, Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
    // Checking features for the account is not supported
    @Override
    public Bundle hasFeatures(
        AccountAuthenticatorResponse r,
        Account account, String[] strings) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
}
```

将授权器绑定到框架

为了让 Sync Adapter 框架可以访问我们的授权器，我们必须为它创建一个绑定服务。这一服务提供一个 Android Binder 对象，允许框架调用我们的授权器，并且在授权器和框架间传递数据。

因为框架会在它第一次需要访问授权器时启动该 Service，所以我们也可以使用该服务来实例化授权器。具体而言，我们需要在服务的 Service.onCreate() 方法中调用授权器的构造函数。

下面的代码样例展示了如何定义绑定 Service：

```
/**
 * A bound Service that instantiates the authenticator
 * when started.
 */
public class AuthenticatorService extends Service {
    ...
    // Instance field that stores the authenticator object
    private Authenticator mAuthenticator;
    @Override
    public void onCreate() {
        // Create a new authenticator object
        mAuthenticator = new Authenticator(this);
    }
    /*
     * When the system binds to this Service to make the RPC call
     * return the authenticator's IBinder.
     */
    @Override
    public IBinder onBind(Intent intent) {
        return mAuthenticator.getIBinder();
    }
}
```

添加授权器的元数据（Metadata）文件

若要将我们的授权器组件集成到 Sync Adapter 框架和账户框架中，我们需要为这些框架提供带有描述组件信息的元数据。该元数据声明了我们为 Sync Adapter 创建的账户类型以及系统所显示的 UI 元素（如果希望用户可以看到我们创建的账户类型）。在我们的项目目录 `/res/xml/` 下，将元数据声明于一个 XML 文件中。我们可以自己为该文件按命名，通常我们将它命名为 `authenticator.xml` 。

在这个 XML 文件中，包含了一个 `<account-authenticator>` 标签，它有下列一些属性：

android:accountType

Sync Adapter 框架要求每一个适配器都有一个域名形式的账户类型。框架会将它作为 Sync Adapter 内部标识的一部分。如果服务端需要登陆，账户类型会和账户一起发送到服务端作为登录凭据的一部分。

如果我们的服务端不需要登录，我们仍然需要提供一个账户类型（该属性的值用我们能控制的一个域名即可）。虽然框架会使用它来管理 Sync Adapter，但该属性的值不会发送到服务端。

android:icon

指向一个包含图标的 Drawable 资源。如果我们在 `res/xml/syncadapter.xml` 中通过指定 `android:userVisible="true"` 让 Sync Adapter 可见，那么我们必须提供图标资源。它会在系统的设置中的账户（Accounts）这一栏内显示。

android:smallIcon

指向一个包含微小版本图标的 Drawable 资源。当屏幕尺寸较小时，这一资源可能会替代 `android:icon` 中所指定的图标资源。

android:label

指明了用户账户类型的本地化字符串。如果我们在 `res/xml/syncadapter.xml` 中通过指定 `android:userVisible="true"` 让 Sync Adapter 可见，那么我们需要提供该字符串。它会在系统的设置中的账户这一栏内显示，就在我们为授权器定义的图标旁边。

下面的代码样例展示了我们之前为授权器创建的 XML 文件：

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:accountType="example.com"
        android:icon="@drawable/ic_launcher"
        android:smallIcon="@drawable/ic_launcher"
        android:label="@string/app_name"/>
```

在 Manifest 文件中声明授权器

在之前的步骤中，我们已经创建了一个绑定服务，将授权器和 Sync Adapter 框架连接了起来。为了让系统可以识别该服务，我们需要在 Manifest 文件中添加 `<service>` 标签，将它作为 `<application>` 的子标签：

```
<service
        android:name="com.example.android.syncadapter.AuthenticatorService">
    <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator"/>
    </intent-filter>
    <meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
</service>
```

`<intent-filter>` 标签配置了一个可以被 `android.accounts.AccountAuthenticator` 这一 Action 所激活的过滤器，这一 Intent 会在系统要运行授权器时由系统发出。当过滤器被激活后，系统会启动 `AuthenticatorService`，即之前用来封装授权器的 Service。

`<meta-data>` 标签声明了授权器的元数据。android:name 属性将元数据和授权器框架连接起来。android:resource 指定了我们之前所创建的授权器元数据文件的名字。

除了授权器之外，Sync Adapter 框架也需要一个 Content Provider。如果我们的应用并没有使用 Content Provider，那么可以阅读下一节课程学习如何创建一个 Stub Content Provider；如果我们的应用已经使用了 ContentProvider，可以直接阅读：创建 Sync Adapter。

创建 Stub Content Provider

> 编写:jdneo - 原文:http://developer.android.com/training/sync-adapters/creating-stub-provider.html

Sync Adapter 框架是设计成用来和设备数据一起工作的，而这些设备数据应该被灵活且安全的 Content Provider 框架管理。因此，Sync Adapter 框架会期望应用已经为它的本地数据定义了 Content Provider。如果 Sync Adapter 框架尝试去运行我们的 Sync Adapter，而我们的应用没有一个 Content Provider 的话，那么 Sync Adapter 将会崩溃。

如果我们正在开发一个新的应用，它将数据从服务器传输到一台设备上，那么我们务必考虑将本地数据存储于 Content Provider 中。除了它对于 Sync Adapter 的重要性之外，Content Provider 还可以提供许多安全上的好处，更何况它是专门为了在 Android 设备上处理数据存储而设计的。要学习如何创建一个 Content Provider，可以阅读：Creating a Content Provider。

然而，如果我们已经通过别的形式来存储本地数据，我们仍然可以使用 Sync Adapter 来处理数据传输。为了满足 Sync Adapter 框架对于 Content Provider 的要求，我们可以在应用中添加一个 Stub Content Provider。一个 Stub Content Provider 实现了 Content Provider 类，但是所有的方法都返回 `null` 或者 `0`。如果我们添加了一个 Stub Content Provider，那么无论数据存储机制是什么，我们都可以使用 Sync Adapter 来传输数据。

如果在我们的应用中已经有了一个 Content Provider，那么我们就不需要创建 Stub Content Provider 了。在这种情况下，我们可以略过这节课程，直接进入：创建 Sync Adapter。如果你还没有创建 Content Provider，这节课将向你展示如何通过添加一个 Stub Content Provider，将你的 Sync Adapter 添加到框架中。

添加一个 Stub Content Provider

要为我们的应用创建一个 Stub Content Provider，首先继承 ContentProvider 类，并且在所有需要重写的方法中，我们一律不进行任何处理而是直接返回。下面的代码片段展示了我们应该如何创建一个 Stub Content Provider：

```
/*
 * Define an implementation of ContentProvider that stubs out
 * all methods
 */
public class StubProvider extends ContentProvider {
    /*
     * Always return true, indicating that the
     * provider loaded correctly.
```

```
     */
    @Override
    public boolean onCreate() {
        return true;
    }
    /*
     * Return an empty String for MIME type
     */
    @Override
    public String getType() {
        return new String();
    }
    /*
     * query() always returns no results
     *
     */
    @Override
    public Cursor query(
            Uri uri,
            String[] projection,
            String selection,
            String[] selectionArgs,
            String sortOrder) {
        return null;
    }
    /*
     * insert() always returns null (no URI)
     */
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }
    /*
     * delete() always returns "no rows affected" (0)
     */
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
    /*
     * update() always returns "no rows affected" (0)
     */
    public int update(
            Uri uri,
            ContentValues values,
            String selection,
            String[] selectionArgs) {
        return 0;
    }
}
```

在 Manifest 清单文件中声明 Provider

Sync Adapter 框架会通过查看应用的 manifest 文件中是否声明了 provider，来验证我们的应用是否使用了 Content Provider。为了在 manifest 清单文件中声明我们的 Stub Content Provider，添加一个 `<provider>` 标签，并让它拥有下列属性字段：

`android:name="com.example.android.datasync.provider.StubProvider"`

指定实现 Stub Content Provider 类的完整包名。

`android:authorities="com.example.android.datasync.provider"`

指定 Stub Content Provider 的 URI Authority。用应用的包名加上字符串 `".provider"` 作为该属性字段的值。虽然我们在这里向系统声明了 Stub Content Provider，但是不会尝试访问 Provider 本身。

`android:exported="false"`

确定其它应用是否可以访问 Content Provider。对于 Stub Content Provider 而言，由于没有让其它应用访问该 Provider 的必要，所以我们将该值设置为 `false`。该值并不会影响 Sync Adapter 框架和 Content Provider 之间的交互。

`android:syncable="true"`

该标识指明 Provider 是可同步的。如果将这个值设置为 `true`，那么将不需要在代码中调用 setIsSyncable()。这一标识将会允许 Sync Adapter 框架和 Content Provider 进行数据传输，但是仅仅在我们显式地执行相关调用时，这一传输时才会进行。

下面的代码片段展示了我们应该如何将 `<provider>` 标签添加到应用的 manifest 清单文件中：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.network.sync.BasicSyncAdapter"
    android:versionCode="1"
    android:versionName="1.0" >
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
    ...
    <provider
        android:name="com.example.android.datasync.provider.StubProvider"
        android:authorities="com.example.android.datasync.provider"
        android:exported="false"
        android:syncable="true"/>
    ...
    </application>
</manifest>
```

现在我们已经创建了所有 Sync Adapter 框架所需要的依赖项，接下来我们可以创建封装数据传输代码的组件了。该组件就叫做 Sync Adapter。在下节课中，我们将会展示如何将这一组件添加到应用中。

创建 Sync Adpater

> 编写:jdneo - 原文:http://developer.android.com/training/sync-adapters/creating-sync-adapter.html

设备和服务器之间执行数据传输的代码会封装在应用的 Sync Adapter 组件中。Sync Adapter 框架会基于我们的调度和触发操作，运行 Sync Adapter 组件中的代码。要将同步适配组件添加到应用当中，我们需要添加下列部件：

Sync Adapter 类

将我们的数据传输代码封装到一个与 Sync Adapter 框架兼容的接口当中。

绑定 Service

通过一个绑定服务，允许 Sync Adapter 框架运行 Sync Adapter 类中的代码。

Sync Adapter 的 XML 元数据文件

该文件包含了有关 Sync Adapter 的信息。框架会根据该文件确定应该如何加载并调度数据传输任务。

应用 manifest 清单文件的声明

需要在应用的 manifest 清单文件中声明绑定服务；同时还需要指出 Sync Adapter 的元数据。

这节课将会向我们展示如何定义他们。

创建一个 Sync Adapter 类

在这部分课程中，我们将会学习如何创建封装了数据传输代码的 Sync Adapter 类。创建该类需要继承 Sync Adapter 的基类；为该类定义构造函数；以及实现相关的方法。在这些方法中，我们定义数据传输任务。

继承 Sync Adapter 基类：AbstractThreadedSyncAdapter

要创建 Sync Adapter 组件，首先继承 AbstractThreadedSyncAdapter，然后编写它的构造函数。与使用 Activity.onCreate() 配置 Activity 时一样，每次我们重新创建 Sync Adapter 组件的时候，使用构造函数执行相关的配置。例如，如果我们的应用使用一个 Content Provider 来存储数据，那么使用构造函数来获取一个 ContentResolver 实例。由于从 Android 3.0 开始添加了第二种形式的构造函数，来支持 `parallelSyncs` 参数，所以我们需要创建两种形式的构造函数来保证兼容性。

> **Note**：Sync Adapter 框架是设计成和 Sync Adapter 组件的单例一起工作的。实例化 Sync Adapter 组件的更多细节，会在后面的章节中展开。

下面的代码展示了如何实现 AbstractThreadedSyncAdapter 和它的构造函数：

```
/**
 * Handle the transfer of data between a server and an
 * app, using the Android sync adapter framework.
 */
public class SyncAdapter extends AbstractThreadedSyncAdapter {
    ...
    // Global variables
    // Define a variable to contain a content resolver instance
    ContentResolver mContentResolver;
    /**
     * Set up the sync adapter
     */
    public SyncAdapter(Context context, boolean autoInitialize) {
        super(context, autoInitialize);
        /*
         * If your app uses a content resolver, get an instance of it
         * from the incoming Context
         */
        mContentResolver = context.getContentResolver();
    }
    ...
    /**
     * Set up the sync adapter. This form of the
     * constructor maintains compatibility with Android 3.0
     * and later platform versions
     */
    public SyncAdapter(
            Context context,
            boolean autoInitialize,
            boolean allowParallelSyncs) {
        super(context, autoInitialize, allowParallelSyncs);
        /*
         * If your app uses a content resolver, get an instance of it
         * from the incoming Context
         */
        mContentResolver = context.getContentResolver();
        ...
    }
```

在 onPerformSync() 中添加数据传输代码

Sync Adapter 组件并不会自动地执行数据传输。它对我们的数据传输代码进行封装，使得 Sync Adapter 框架可以在后台执行数据传输，而不会牵连到我们的应用。当框架准备同步我们的应用数据时，它会调用我们所实现的 onPerformSync() 方法。

为了便于将数据从应用程序转移到 Sync Adapter 组件中，Sync Adapter 框架调用 onPerformSync()，它具有下面的参数：

Account

该 Account 对象与触发 Sync Adapter 的事件相关联。如果服务端不需要使用账户，那么我们不需要使用这个对象内的信息。

Extras

一个 Bundle 对象，它包含了一些标识，这些标识由触发 Sync Adapter 的事件所发送。

Authority

系统中某个 Content Provider 的 Authority。我们的应用必须要有访问它的权限。通常，该 Authority 对应于应用的 Content Provider。

Content provider client

ContentProviderClient 针对于由 `Authority` 参数所指向的Content Provider。ContentProviderClient 是一个 Content Provider 的轻量级共有接口。它的基本功能和 ContentResolver 一样。如果我们正在使用 Content Provider 来存储应用数据，那么我们可以利用它连接 Content Provider。反之，则将其忽略。

Sync result

一个 SyncResult 对象，我们可以使用它将信息发送给 Sync Adapter 框架。

下面的代码片段展示了 onPerformSync() 函数的整体结构：

```
/*
 * Specify the code you want to run in the sync adapter. The entire
 * sync adapter runs in a background thread, so you don't have to set
 * up your own background processing.
 */
@Override
public void onPerformSync(
        Account account,
        Bundle extras,
        String authority,
        ContentProviderClient provider,
        SyncResult syncResult) {
/*
 * Put the data transfer code here.
 */
...
}
```

虽然实际的onPerformSync() 实现是要根据应用数据的同步需求以及服务器的连接协议来制定，但是我们的实现只需要执行一些常规任务：

连接到一个服务器

尽管我们可以假定在开始传输数据时，已经获取到了网络连接，但是 Sync Adapter 框架并不会自动地连接到一个服务器。

下载和上传数据

Sync Adapter 不会自动执行数据传输。如果我们想要从服务器下载数据并将它存储到 Content Provider 中，我们必须提供请求数据，下载数据和将数据插入到 Provider 中的代码。类似地，如果我们想把数据发送到服务器，我们需要从一个文件，数据库或者 Provider 中读取数据，并且发送必需的上传请求。同时我们还需要处理在执行数据传输时所发生的网络错误。

处理数据冲突或者确定当前数据的状态

Sync Adapter 不会自动地解决服务器数据与设备数据之间的冲突。同时，它也不会自动检测服务器上的数据是否比设备上的数据要新，反之亦然。因此，我们必须自己提供处理这些状况的算法。

清理

在数据传输的尾声，记得要关闭网络连接，清除临时文件和缓存。

> **Note**：Sync Adapter 框架会在一个后台线程中执行 onPerformSync() 方法，所以我们不需要配置后台处理任务。

除了和同步相关的任务之外，我们还应该尝试将一些周期性的网络相关的任务合并起来，并将它们添加到 onPerformSync() 中。将所有网络任务集中到该方法内处理，可以减少由启动和停止网络接口所造成的电量损失。有关更多如何在进行网络访问时更高效地使用电池方面的知识，可以阅读：Transferring Data Without Draining the Battery，它描述了一些在数据传输代码中可以包含的网络访问任务。

将 Sync Adapter 绑定到框架上

现在，我们已经将数据传输代码封装在 Sync Adapter 组件中，但是我们必须让框架可以访问我们的代码。为了做到这一点，我们需要创建一个绑定 Service，它将一个特殊的 Android Binder 对象从 Sync Adapter 组件传递给框架。有了这一 Binder 对象，框架就可以调用 onPerformSync() 方法并将数据传递给它。

在服务的 onCreate() 方法中将我们的 Sync Adapter 组件实例化为一个单例。通过在 onCreate() 方法中实例化该组件，我们可以推迟到服务启动后再创建它，这会在框架第一次尝试执行数据传输时发生。我们需要通过一种线程安全的方法来实例化组件，以防止 Sync

Adapter 框架在响应触发和调度时，形成含有多个 Sync Adapter 执行的队列。

下面的代码片段展示了我们应该如何实现一个绑定 Service 的类，实例化我们的 Sync Adapter 组件，并获取 Android Binder 对象：

```
package com.example.android.syncadapter;
/**
 * Define a Service that returns an IBinder for the
 * sync adapter class, allowing the sync adapter framework to call
 * onPerformSync().
 */
public class SyncService extends Service {
    // Storage for an instance of the sync adapter
    private static SyncAdapter sSyncAdapter = null;
    // Object to use as a thread-safe lock
    private static final Object sSyncAdapterLock = new Object();
    /*
     * Instantiate the sync adapter object.
     */
    @Override
    public void onCreate() {
        /*
         * Create the sync adapter as a singleton.
         * Set the sync adapter as syncable
         * Disallow parallel syncs
         */
        synchronized (sSyncAdapterLock) {
            if (sSyncAdapter == null) {
                sSyncAdapter = new SyncAdapter(getApplicationContext(), true);
            }
        }
    }
    /**
     * Return an object that allows the system to invoke
     * the sync adapter.
     *
     */
    @Override
    public IBinder onBind(Intent intent) {
        /*
         * Get the object that allows external processes
         * to call onPerformSync(). The object is created
         * in the base class code when the SyncAdapter
         * constructors call super()
         */
        return sSyncAdapter.getSyncAdapterBinder();
    }
}
```

> **Note**：要看更多 Sync Adapter 绑定服务的例子，可以阅读样例代码。

添加框架所需的账户

Sync Adapter 框架需要每个 Sync Adapter 拥有一个账户类型。在创建 Stub 授权器章节中，我们已经声明了账户类型的值。现在我们需要在 Android 系统中配置该账户类型。要配置账户类型，通过调用 addAccountExplicitly() 添加一个使用其账户类型的虚拟账户。

调用该方法最合适的地方是在应用的启动 Activity 的 onCreate() 方法中。如下面的代码样例所示：

```
public class MainActivity extends FragmentActivity {
    ...
    ...
    // Constants
    // The authority for the sync adapter's content provider
    public static final String AUTHORITY = "com.example.android.datasync.provider"
    // An account type, in the form of a domain name
    public static final String ACCOUNT_TYPE = "example.com";
    // The account name
    public static final String ACCOUNT = "dummyaccount";
    // Instance fields
    Account mAccount;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Create the dummy account
        mAccount = CreateSyncAccount(this);
        ...
    }
    ...
    /**
     * Create a new dummy account for the sync adapter
     *
     * @param context The application context
     */
    public static Account CreateSyncAccount(Context context) {
        // Create the account type and default account
        Account newAccount = new Account(
                ACCOUNT, ACCOUNT_TYPE);
        // Get an instance of the Android account manager
        AccountManager accountManager =
                (AccountManager) context.getSystemService(
                        ACCOUNT_SERVICE);
        /*
         * Add the account and account type, no password or user data
         * If successful, return the Account object, otherwise report an error.
         */
        if (accountManager.addAccountExplicitly(newAccount, null, null))) {
            /*
```

```
                * If you don't set android:syncable="true" in
                * in your <provider> element in the manifest,
                * then call context.setIsSyncable(account, AUTHORITY, 1)
                * here.
                */
        } else {
            /*
                * The account exists or some other error occurred. Log this, report it,
                * or handle it internally.
                */
        }
    }
    ...
}
```

添加 Sync Adapter 的元数据文件

要将我们的 Sync Adapter 组件集成到框架中，我们需要向框架提供描述组件的元数据，以及额外的标识信息。元数据指定了我们为 Sync Adapter 所创建的账户类型，声明了一个和应用相关联的 Content Provider Authority，对和 Sync Adapter 相关的一部分系统用户接口进行控制，同时还声明了其它同步相关的标识。在我们项目的 `/res/xml/` 目录下的一个特定文件内声明这一元数据，我们可以为这个文件命名，不过通常来说我们将其命名为 `syncadapter.xml` 。

在这一文件中包含了一个 XML 标签 `<sync-adapter>`，它包含了下列的属性字段：

`android:contentAuthority`

Content Provider 的 URI Authority。如果我们在前一节课程中为应用创建了一个 Stub Content Provider，那么请使用在 manifest 清单文件中添加在 `<provider>` 标签内的 android:authorities 属性值。这一属性的更多细节在本章后续章节中有更多的介绍。

如果我们正使用 Sync Adapter 将数据从 Content Provider 传输到服务器上，该属性的值应该和数据的 Content URI Authority 保持一致。这个值也是我们在 manifest 清单文件中添加在 `<provider>` 标签内 `android:authorities` 属性的值。

`android:accountType`

Sync Adapter 框架所需要的账户类型。这个值必须和我们所创建的验证器元数据文件内所提供的账户类型一致（详细内容可以阅读：创建 Stub 授权器）。这也是在上一节的代码片段中。常量 `ACCOUNT_TYPE` 的值。

配置相关属性

`android:userVisible`

该属性设置 Sync Adapter 框架的账户类型是否可见。默认地，和账户类型相关联的账户图标和标签在系统设置的账户选项中可以看见，所以我们应该将 Sync Adapter 设置为对用户不可见（除非我们确实拥有一个账户类型或者域名或者它们可以轻松地和我们的应用相关联）。如果我们将账户类型设置为不可见，那么我们仍然可以允许用户通过一个 Activity 中的用户接口来控制 Sync Adapter。

`android:supportsUploading`

允许我们将数据上传到云。如果应用仅仅下载数据，那么请将该属性设置为 `false` 。

`android:allowParallelSyncs`

允许多个 Sync Adapter 组件的实例同时运行。如果应用支持多个用户账户并且我们希望多个用户并行地传输数据，那么可以使用该特性。如果我们从不执行多个数据传输，那么这个选项是没用的。

`android:isAlwaysSyncable`

指明 Sync Adapter 框架可以在任何我们指定的时间运行 Sync Adapter。如果我们希望通过代码来控制 Sync Adapter 的运行时机，请将该属性设置为 `false` 。然后调用 requestSync() 来运行 Sync Adapter。要学习更多关于运行 Sync Adapter 的知识，可以阅读：执行 Sync Adapter。

下面的代码展示了应该如何通过 XML 配置一个使用单个虚拟账户，并且只执行下载的 Sync Adapter：

```xml
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:contentAuthority="com.example.android.datasync.provider"
        android:accountType="com.android.example.datasync"
        android:userVisible="false"
        android:supportsUploading="false"
        android:allowParallelSyncs="false"
        android:isAlwaysSyncable="true"/>
```

在 Manifest 清单文件中声明 Sync Adapter

一旦我们将 Sync Adapter 组件集成到应用中，我们需要声明相关的权限来使用它，并且还需要声明我们所添加的绑定 Service。

由于 Sync Adapter 组件会运行设备与网络之间传输数据的代码，所以我们需要请求使用网络的权限。同时，我们的应用还需要读写 Sync Adapter 配置信息的权限，这样我们才能通过应用中的其它组件去控制 Sync Adapter。另外，我们还需要一个特殊的权限，来允许应用使用我们在创建 Stub 授权器中所创建的授权器组件。

要请求这些权限，将下列内容添加到应用 manifest 清单文件中，并作为 `<manifest>` 标签的子标签：

`android.permission.INTERNET`

允许 Sync Adapter 访问网络，使得它可以从设备下载和上传数据到服务器。如果之前已经请求了该权限，那么就不需要重复请求了。

`android.permission.READ_SYNC_SETTINGS`

允许应用读取当前的 Sync Adapter 配置。例如，我们需要该权限来调用 getIsSyncable()。

`android.permission.WRITE_SYNC_SETTINGS`

允许我们的应用 对Sync Adapter 的配置进行控制。我们需要这一权限来通过 addPeriodicSync() 方法设置执行同步的时间间隔。另外，调用 requestSync() 方法不需要用到该权限。更多信息可以阅读：执行 Sync Adapter。

`android.permission.AUTHENTICATE_ACCOUNTS`

允许我们使用在创建 Stub 授权器中所创建的验证器组件。

下面的代码片段展示了如何添加这些权限：

```
<manifest>
...
    <uses-permission
            android:name="android.permission.INTERNET"/>
    <uses-permission
            android:name="android.permission.READ_SYNC_SETTINGS"/>
    <uses-permission
            android:name="android.permission.WRITE_SYNC_SETTINGS"/>
    <uses-permission
            android:name="android.permission.AUTHENTICATE_ACCOUNTS"/>
...
</manifest>
```

最后，要声明框架用来和 Sync Adapter 进行交互的绑定 Service，添加下列的 XML 代码到应用 manifest 清单文件中，作为 `<application>` 标签的子标签：

```
        <service
                android:name="com.example.android.datasync.SyncService"
                android:exported="true"
                android:process=":sync">
            <intent-filter>
                <action android:name="android.content.SyncAdapter"/>
            </intent-filter>
            <meta-data android:name="android.content.SyncAdapter"
                    android:resource="@xml/syncadapter" />
        </service>
```

`<intent-filter>` 标签配置了一个过滤器，它会被带有 `android.content.SyncAdapter` 这一 Action 的 Intent 所触发，该 Intent 一般是由系统为了运行 Sync Adapter 而发出的。当过滤器被触发后，系统会启动我们所创建的绑定服务，在本例中它叫做 `SyncService` 。属性 android:exported="true" 允许我们应用之外的其它进程（包括系统）访问这一 Service。属性 android:process=":sync" 告诉系统应该在一个全局共享的，且名字叫做 `sync` 的进程内运行该 Service。如果我们的应用中有多个 Sync Adapter，那么它们可以共享该进程，这有助于减少开销。

`<meta-data>` 标签提供了我们之前为 Sync Adapter 所创建的元数据 XML 文件的文件名。属性 android:name 指出这一元数据是针对于 Sync Adapter 框架的。而 android:resource 标签则指定了元数据文件的名称。

现在我们已经为 Sync Adapter 准备好所有相关的组件了。下一节课将讲授如何让 Sync Adapter 框架运行 Sync Adapter。要实现这一点，既可以通过响应一个事件的方式，也可以通过执行一个周期性任务的方式。

执行 Sync Adpater

> 编写:jdneo - 原文:http://developer.android.com/training/sync-adapters/running-sync-adapter.html

在本节课之前，我们已经学习了如何创建一个封装了数据传输代码的 Sync Adapter 组件，以及如何添加其它的组件，使得我们可以将 Sync Adapter 集成到系统当中。现在我们已经拥有了所有部件，来安装一个包含有 Sync Adapter 的应用了，但是这里还没有任何代码是负责去运行 Sync Adapter。

执行 Sync Adapter 的时机，一般应该基于某个计划任务或者一些事件的间接结果。例如，我们可能希望 Sync Adapter 以一个定期计划任务的形式运行（比如每隔一段时间或者在每天的一个固定时间运行）。或者也可能希望当设备上的数据发生变化后，执行 Sync Adapter。我们应该避免将运行 Sync Adapter 作为用户某个行为的直接结果，因为这样做的话我们就无法利用 Sync Adapter 框架可以按计划调度的特性。例如，我们应该在 UI 中避免使用刷新按钮。

下列情况可以作为运行 Sync Adapter 的时机：

当服务端数据变更时：

当服务端发送消息告知服务端数据发生变化时，运行 Sync Adapter 以响应这一来自服务端的消息。这一选项允许从服务器更新数据到设备上，该方法可以避免由于轮询服务器所造成的执行效率下降，或者电量损耗。

当设备的数据变更时：

当设备上的数据发生变化时，运行 Sync Adapter。这一选项允许我们将修改后的数据从设备发送给服务器。如果需要保证服务器端一直拥有设备上最新的数据，那么这一选项非常有用。如果我们将数据存储于 Content Provider，那么这一选项的实现将会非常直接。如果使用的是一个 Stub Content Provider，检测数据的变化可能会比较困难。

当系统发送了一个网络消息：

当 Android 系统发送了一个网络消息来保持 TCP/IP 连接开启时，运行 Sync Adapter。这个消息是网络框架（Networking Framework）的一个基本部分。可以将这一选项作为自动运行 Sync Adapter 的一个方法。另外还可以考虑将它和基于时间间隔运行 Sync Adapter 的策略结合起来使用。

每隔一定时间：

可以每隔一段指定的时间间隔后，运行 Sync Adapter，或者在每天的固定时间运行它。

根据需求：

运行 Sync Adapter 以响应用户的行为。然而，为了提供最佳的用户体验，我们应该主要依赖那些更加自动式的选项。使用自动式的选项，可以节省大量的电量以及网络资源。

本课程的后续部分会详细介绍每个选项。

当服务器数据变化时，运行 Sync Adapter

如果我们的应用从服务器传输数据，且服务器的数据会频繁地发生变化，那么可以使用一个 Sync Adapter 通过下载数据来响应服务端数据的变化。要运行 Sync Adapter，我们需要让服务端向应用的 BroadcastReceiver 发送一条特殊的消息。为了响应这条消息，可以调用 ContentResolver.requestSync() 方法，向 Sync Adapter 框架发出信号，让它运行 Sync Adapter。

谷歌云消息（Google Cloud Messaging，GCM）提供了我们需要的服务端组件和设备端组件，来让上述消息系统能够运行。使用 GCM 触发数据传输比通过向服务器轮询的方式要更加可靠，也更加有效。因为轮询需要一个一直处于活跃状态的 Service，而 GCM 使用的 BroadcastReceiver 仅在消息到达时会被激活。另外，即使没有更新的内容，定期的轮询也会消耗大量的电池电量，而 GCM 仅在需要时才会发出消息。

> **Note**：如果我们使用 GCM，将广播消息发送到所有安装了我们的应用的设备，来激活 Sync Adapter。要记住他们会在同一时间（粗略地）收到我们的消息。这会导致在同一时段内有多个 Sync Adapter 的实例在运行，进而导致服务器和网络的负载过重。要避免这一情况，我们应该考虑为不同的设备设定不同的 Sync Adapter 来延迟启动时间。

下面的代码展示了如何通过 requestSync() 响应一个接收到的 GCM 消息：

```
public class GcmBroadcastReceiver extends BroadcastReceiver {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY = "com.example.android.datasync.provider"
    // Account type
    public static final String ACCOUNT_TYPE = "com.example.android.datasync";
    // Account
    public static final String ACCOUNT = "default_account";
    // Incoming Intent key for extended data
    public static final String KEY_SYNC_REQUEST =
            "com.example.android.datasync.KEY_SYNC_REQUEST";
    ...
    @Override
    public void onReceive(Context context, Intent intent) {
        // Get a GCM object instance
        GoogleCloudMessaging gcm =
                GoogleCloudMessaging.getInstance(context);
        // Get the type of GCM message
        String messageType = gcm.getMessageType(intent);
        /*
         * Test the message type and examine the message contents.
         * Since GCM is a general-purpose messaging system, you
         * may receive normal messages that don't require a sync
         * adapter run.
         * The following code tests for a a boolean flag indicating
         * that the message is requesting a transfer from the device.
         */
        if (GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType)
            &&
            intent.getBooleanExtra(KEY_SYNC_REQUEST)) {
            /*
             * Signal the framework to run your sync adapter. Assume that
             * app initialization has already created the account.
             */
            ContentResolver.requestSync(ACCOUNT, AUTHORITY, null);
            ...
        }
        ...
    }
    ...
}
```

当 Content Provider 的数据变化时，运行 Sync Adapter

如果我们的应用在一个 Content Provider 中收集数据，并且希望当我们更新了 Content Provider 的时候，同时更新服务器的数据，我们可以配置 Sync Adapter 来让它自动运行。要做到这一点，首先应该为 Content Provider 注册一个 Observer。当 Content Provider 的数据发生了变化之后，Content Provider 框架会调用 Observer。在 Observer 中，调用 requestSync() 来告诉框架现在应该运行 Sync Adapter 了。

> **Note**：如果我们使用的是一个 Stub Content Provider，那么在 Content Provider 中不会有任何数据，并且不会调用 onChange() 方法。在这种情况下，我们不得不提供自己的某种机制来检测设备数据的变化。这一机制还要负责在数据发生变化时调用 requestSync()。

为了给 Content Provider 创建一个 Observer，继承 ContentObserver 类，并且实现 onChange() 方法的两种形式。在 onChange() 中，调用 requestSync() 来启动 Sync Adapter。

要注册 Observer，需要将它作为参数传递给 registerContentObserver()。在该方法中，我们还要传递一个我们想要监视的 Content URI。Content Provider 框架会将这个需要监视的 URI 与其它一些 Content URIs 进行比较，这些其它的 Content URIs 来自于 ContentResolver 中那些可以修改 Provider 的方法（如 ContentResolver.insert()）所传入的参数。如果出现了变化，那么我们所实现的 ContentObserver.onChange() 将会被调用。

下面的代码片段展示了如何定义一个 ContentObserver，它在表数据发生变化后调用 requestSync()：

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider scheme
    public static final String SCHEME = "content://";
    // Content provider authority
    public static final String AUTHORITY = "com.example.android.datasync.provider";
    // Path for the content provider table
    public static final String TABLE_PATH = "data_table";
    // Account
    public static final String ACCOUNT = "default_account";
    // Global variables
    // A content URI for the content provider's data table
    Uri mUri;
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    public class TableObserver extends ContentObserver {
        /*
         * Define a method that's called when data in the
         * observed content provider changes.
         * This method signature is provided for compatibility with
         * older platforms.
         */
```

```
        @Override
        public void onChange(boolean selfChange) {
            /*
             * Invoke the method signature available as of
             * Android platform version 4.1, with a null URI.
             */
            onChange(selfChange, null);
        }
        /*
         * Define a method that's called when data in the
         * observed content provider changes.
         */
        @Override
        public void onChange(boolean selfChange, Uri changeUri) {
            /*
             * Ask the framework to run your sync adapter.
             * To maintain backward compatibility, assume that
             * changeUri is null.
            ContentResolver.requestSync(ACCOUNT, AUTHORITY, null);
        }
        ...
    }
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver object for your app
        mResolver = getContentResolver();
        // Construct a URI that points to the content provider data table
        mUri = new Uri.Builder()
                  .scheme(SCHEME)
                  .authority(AUTHORITY)
                  .path(TABLE_PATH)
                  .build();
        /*
         * Create a content observer object.
         * Its code does not mutate the provider, so set
         * selfChange to "false"
         */
        TableObserver observer = new TableObserver(false);
        /*
         * Register the observer for the data table. The table's path
         * and any of its subpaths trigger the observer.
         */
        mResolver.registerContentObserver(mUri, true, observer);
        ...
    }
    ...
}
```

在一个网络消息之后，运行 Sync Adapter

当可以获得一个网络连接时，Android 系统会每隔几秒发送一条消息来保持 TCP/IP 连接处于开启状态。这一消息也会传递到每个应用的 ContentResolver 中。通过调用 setSyncAutomatically()，我们可以在 ContentResolver 收到消息后，运行 Sync Adapter。

每当网络消息被发送后运行 Sync Adapter，通过这样的调度方式可以保证每次运行 Sync Adapter 时都可以访问网络。如果不是每次数据变化时就要以数据传输来响应，但是又希望自己的数据会被定期地更新，那么我们可以用这一选项。类似地，如果我们不想要定期执行 Sync Adapter，但希望经常运行它，我们也可以使用这一选项。

由于 setSyncAutomatically() 方法不会禁用 addPeriodicSync()，所以 Sync Adapter 可能会在一小段时间内重复地被触发激活。如果我们想要定期地运行 Sync Adapter，应该禁用 setSyncAutomatically()。

下面的代码片段展示如何配置 ContentResolver，利用它来响应网络消息，从而运行 Sync Adapter：

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY = "com.example.android.datasync.provider";
    // Account
    public static final String ACCOUNT = "default_account";
    // Global variables
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver for your app
        mResolver = getContentResolver();
        // Turn on automatic syncing for the default account and authority
        mResolver.setSyncAutomatically(ACCOUNT, AUTHORITY, true);
        ...
    }
    ...
}
```

定期地运行Sync Adapter

我们可以设置一个在运行之间的时间间隔来定期运行 Sync Adapter，或者在每天的固定时间运行它，还可以两种策略同时使用。定期地运行 Sync Adapter 可以让服务器的更新间隔大致保持一致。

同样地，当服务器相对来说比较空闲时，我们可以通过在夜间定期调用 Sync Adapter，把设备上的数据上传到服务器。大多数用户在晚上不会关机，并为手机充电，所以这一方法是可行的。而且，通常来说，设备不会在深夜运行除了 Sync Adapter 之外的其他的任务。然而，如果我们使用这个方法的话，我们需要注意让每台设备在略微不同的时间触发数据传输。如果所有设备在同一时间运行我们的 Sync Adapter，那么我们的服务器和移动运营商的网络将很有可能负载过重。

一般来说，当我们的用户不需要实时更新，而希望定期更新时，使用定期运行的策略会很有用。如果我们希望在数据的实时性和 Sync Adapter 的资源消耗之间进行一个平衡，那么定期执行是一个不错的选择。

要定期运行我们的 Sync Adapter，调用 addPeriodicSync()。这样每隔一段时间，Sync Adapter 就会运行。由于 Sync Adapter 框架会考虑其他 Sync Adapter 的执行，并尝试最大化电池效率，所以间隔时间会动态地进行细微调整。同时，如果当前无法获得网络连接，框架不会运行我们的 Sync Adapter。

注意，addPeriodicSync() 方法不会让 Sync Adapter 每天在某个时间自动运行。要让我们的 Sync Adapter 在每天的某个时刻自动执行，可以使用一个重复计时器作为触发器。重复计时器的更多细节可以阅读：AlarmManager。如果我们使用 setInexactRepeating() 方法设置了一个每天的触发时刻会有粗略变化的触发器，我们仍然应该将不同设备 Sync Adapter 的运行时间随机化，使得它们的执行交错开来。

addPeriodicSync() 方法不会禁用 setSyncAutomatically()，所以我们可能会在一小段时间内产生多个 Sync Adapter 的运行实例。另外，仅有一部分 Sync Adapter 的控制标识可以在调用 addPeriodicSync() 时使用。不被允许的标识在该方法的文档中可以查看。

下面的代码样例展示了如何定期执行 Sync Adapter：

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY = "com.example.android.datasync.provider";
    // Account
    public static final String ACCOUNT = "default_account";
    // Sync interval constants
    public static final long SECONDS_PER_MINUTE = 60L;
    public static final long SYNC_INTERVAL_IN_MINUTES = 60L;
    public static final long SYNC_INTERVAL =
            SYNC_INTERVAL_IN_MINUTES *
            SECONDS_PER_MINUTE;
    // Global variables
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver for your app
        mResolver = getContentResolver();
        /*
         * Turn on periodic syncing
         */
        ContentResolver.addPeriodicSync(
                ACCOUNT,
                AUTHORITY,
                Bundle.EMPTY,
                SYNC_INTERVAL);
        ...
    }
    ...
}
```

按需求执行 Sync Adapter

以响应用户请求的方式运行 Sync Adapter 是最不推荐的策略。要知道，该框架是被特别设计的，它可以让 Sync Adapter 在根据某个调度规则运行时，能够尽量最高效地使用手机电量。显然，在数据改变的时候执行同步可以更有效的使用手机电量，因为电量都消耗在了更新新的数据上。

相比之下，允许用户按照自己的需求运行 Sync Adapter 意味着 Sync Adapter 会自己运行，这将无法有效地使用电量和网络资源。如果根据需求执行同步，会诱导用户即便没有证据表明数据发生了变化也请求一个更新，这些无用的更新会导致对电量的低效率使用。一般来

说，我们的应用应该使用其它信号来触发一个同步更新或者让它们定期地去执行，而不是依赖于用户的输入。

不过，如果我们仍然想要按照需求运行 Sync Adapter，可以将 Sync Adapter 的配置标识设置为手动执行，之后调用 ContentResolver.requestSync() 来触发一次更新。

通过下列标识来执行按需求的数据传输：

`SYNC_EXTRAS_MANUAL`

强制执行手动的同步更新。Sync Adapter 框架会忽略当前的设置，比如通过 setSyncAutomatically() 方法设置的标识。

`SYNC_EXTRAS_EXPEDITED`

强制同步立即执行。如果我们不设置此项，系统可能会在运行同步请求之前等待一小段时间，因为它会尝试将一小段时间内的多个请求集中在一起调度，目的是为了优化电量的使用。

下面的代码片段将展示如何调用 requestSync() 来响应一个按钮点击事件：

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY =
            "com.example.android.datasync.provider"
    // Account type
    public static final String ACCOUNT_TYPE = "com.example.android.datasync";
    // Account
    public static final String ACCOUNT = "default_account";
    // Instance fields
    Account mAccount;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        /*
         * Create the dummy account. The code for CreateSyncAccount
         * is listed in the lesson Creating a Sync Adapter
         */

        mAccount = CreateSyncAccount(this);
        ...
    }
    /**
     * Respond to a button click by calling requestSync(). This is an
     * asynchronous operation.
     *
     * This method is attached to the refresh button in the layout
     * XML file
     *
     * @param v The View associated with the method call,
     * in this case a Button
     */
    public void onRefreshButtonClick(View v) {
        ...
        // Pass the settings flags by inserting them in a bundle
        Bundle settingsBundle = new Bundle();
        settingsBundle.putBoolean(
                ContentResolver.SYNC_EXTRAS_MANUAL, true);
        settingsBundle.putBoolean(
                ContentResolver.SYNC_EXTRAS_EXPEDITED, true);
        /*
         * Request the sync for the default account, authority, and
         * manual sync settings
         */
        ContentResolver.requestSync(mAccount, AUTHORITY, settingsBundle);
    }
```

使用 Volley 传输网络数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/volley/index.html

`Volley` 是一个 HTTP 库，它能够帮助 Android app 更方便地执行网络操作，最重要的是，它更快速高效。我们可以通过开源的 AOSP 仓库获取到 Volley 。

YOU SHOULD ALSO SEE

使用 Volley 来编写一个 app，请参考2013 Google I/O schedule app。另外需要特别关注下面2个部分：

- ImageLoader
- BitmapCache

VIDEO - Volley: Easy,Fast Networking for Android

Volley 有如下的优点：

- 自动调度网络请求。
- 高并发网络连接。
- 通过标准的 HTTP cache coherence（高速缓存一致性）缓存磁盘和内存透明的响应。
- 支持指定请求的优先级。
- 撤销请求 API。我们可以取消单个请求，或者指定取消请求队列中的一个区域。
- 框架容易被定制，例如，定制重试或者回退功能。
- 强大的指令（Strong ordering）可以使得异步加载网络数据并正确地显示到 UI 的操作更加简单。
- 包含了调试与追踪工具。

Volley 擅长执行用来显示 UI 的 RPC 类型操作，例如获取搜索结果的数据。它轻松的整合了任何协议，并输出操作结果的数据，可以是原始的字符串，也可以是图片，或者是 JSON。通过提供内置的我们可能使用到的功能，Volley 可以使得我们免去重复编写样板代码，使我们可以把关注点放在 app 的功能逻辑上。

Volley 不适合用来下载大的数据文件。因为 Volley 会保持在解析的过程中所有的响应。对于下载大量的数据操作，请考虑使用 DownloadManager。

Volley 框架的核心代码是托管在 AOSP 仓库的 `frameworks/volley` 中，相关的工具放在 `toolbox` 下。把 Volley 添加到项目中最简便的方法是 Clone 仓库，然后把它设置为一个 library project：

1. 通过下面的命令来Clone仓库：

```
git clone https://android.googlesource.com/platform/frameworks/volley
```

2. 以一个 Android library project 的方式导入下载的源代码到你的项目中。(如果你使用 Eclipse，请参考 Managing Projects from Eclipse with ADT，或者编译成一个 `.jar` 文件。

Lessons

发送一个简单的网络请求(Sending a Simple Request)

学习如何通过 Volley 默认的行为发送一个简单的请求，以及如何取消一个请求。

建立一个请求队列(Setting Up a RequestQueue)

学习如何建立一个请求队列（`RequestQueue`），以及如何实现一个单例模式来创建一个请求队列，使 `RequestQueue` 能够持续保持在我们 app 的生命周期中。

生成一个标准的请求(Making a Standard Request)

学习如何使用 Volley 的 out-of-the-box（可直接使用、无需配置）请求类型（原始字符串、图片和 JSON）来发送一个请求。

实现自定义的请求(Implementing a Custom Request)

学习如何实现一个自定义的请求。

发送简单的网络请求

> 编写:kesenhoo - 原文:http://developer.android.com/training/volley/simple.html

使用 Volley 的方式是，创建一个 `RequestQueue` 并传递 `Request` 对象给它。`RequestQueue` 管理用来执行网络操作的工作线程，从缓存中读取数据，写数据到缓存，并解析 Http 的响应内容。请求解析原始的响应数据，Volley 会把解析完的响应数据分发给主线程。

这节课会介绍如何使用 `Volley.newRequestQueue` 这个便捷的方法（建立一个请求队列 `RequestQueue` ）来发送一个请求。在下一节课建立一个 RequestQueue中，会介绍如何自己建立一个 `RequestQueue` 。

这节课也会介绍如何添加一个请求到 `RequesutQueue` 以及如何取消一个请求。

1)Add the INTERNET Permission

为了使用Volley，你必须添加`android.permission.INTERNET`权限到你的manifest文件中。没有这个权限，你的app将无法访问网络。

2)Use newRequestQueue

Volley提供了一个简便的方法：`Volley.newRequestQueue`用来为你建立一个`RequestQueue`，使用默认值，并启动这个队列。例如：

```
final TextView mTextView = (TextView) findViewById(R.id.text);
...

// Instantiate the RequestQueue.
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";

// Request a string response from the provided URL.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
            new Response.Listener() {
    @Override
    public void onResponse(String response) {
        // Display the first 500 characters of the response string.
        mTextView.setText("Response is: "+ response.substring(0,500));
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        mTextView.setText("That didn't work!");
    }
});
// Add the request to the RequestQueue.
queue.add(stringRequest);
```

Volley总是将解析后的数据返回至主线程中。在主线程中更加合适使用接收到的数据用来操作UI控件，这样你可以在响应的handler中轻松的修改UI，但是对于库提供的一些其他方法是有些特殊的，例如与取消有关的。

关于如何创建你自己的请求队列，而不是使用Volley.newRequestQueue方法，请查看建立一个请求队列Setting Up a RequestQueue。

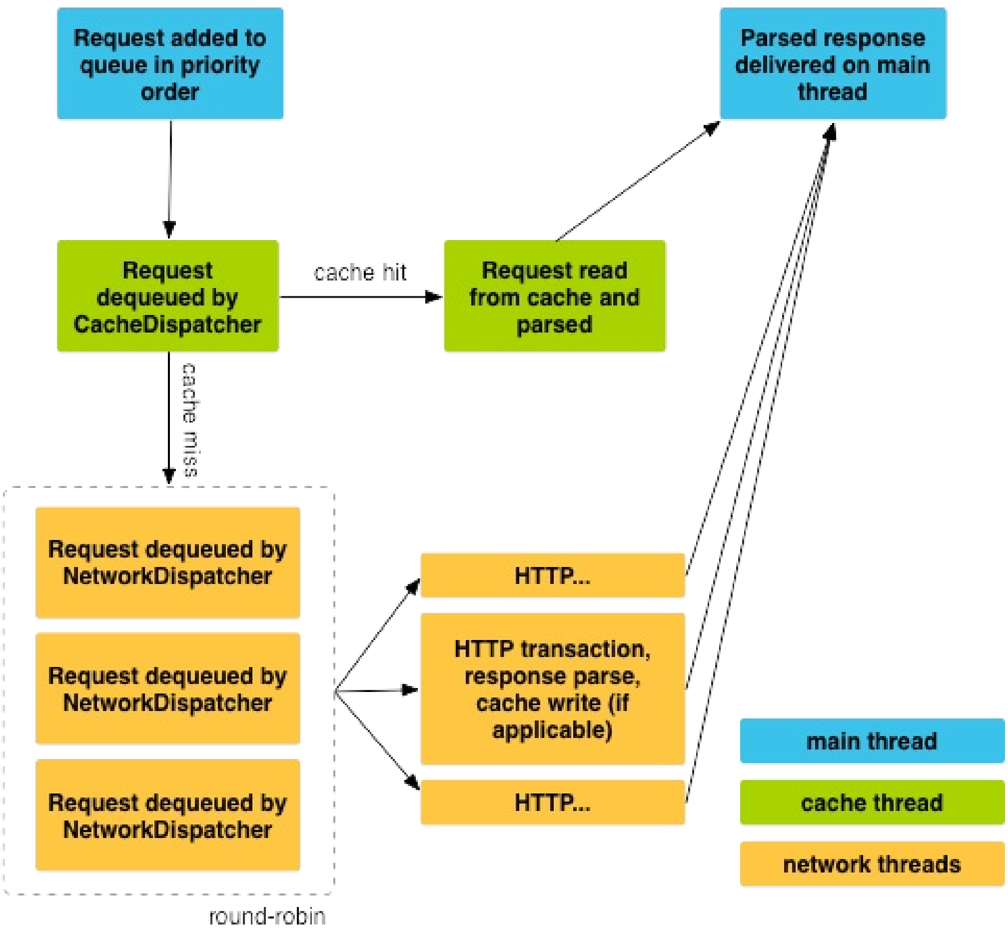
3)Send a Request

为了发送一个请求，你只需要构造一个请求并通过 `add()` 方法添加到 `RequestQueue` 中。一旦你添加了这个请求，它会通过队列，得到处理，然后得到原始的响应数据并返回。

当你执行 `add()` 方法时，Volley触发执行一个缓存处理线程以及一系列网络处理线程。当你添加一个请求到队列中，它将被缓存线程所捕获并触发：如果这个请求可以被缓存处理，那么会在缓存线程中执行响应数据的解析并返回到主线程。如果请求不能被缓存所处理，它会被放到网络队列中。网络线程池中的第一个可用的网络线程会从队列中获取到这个请求并执行HTTP操作，解析工作线程的响应数据，把数据写到缓存中并把解析之后的数据返回到主线程。

请注意那些比较耗时的操作，例如I/O与解析parsing/decoding都是执行在工作线程。你可以在任何线程中添加一个请求，但是响应结果都是返回到主线程的。

下图1，演示了一个请求的生命周期：



4)Cancel a Request

对请求Request对象调用 `cancel()` 方法取消一个请求。一旦取消，Volley会确保你的响应Handler不会被执行。这意味着在实际操作中你可以在activity的 `onStop()` 方法中取消所有pending在队列中的请求。你不需要通过检测 `getActivity() == null` 来丢弃你的响应handler，其他类似 `onSaveInstanceState()` 等保护性的方法里面也都不需要检测。

为了利用这种优势，你应该跟踪所有已经发送的请求，以便在需要的时候可以取消他们。有一个简便的方法：你可以为每一个请求对象都绑定一个tag对象。然后你可以使用这个tag来提供取消的范围。例如，你可以为你的所有请求都绑定到执行的Activity上，然后你可以在 `onStop()` 方法执行 `requestQueue.cancelAll(this)` 。同样的，你可以为ViewPager中的所有请求缩略图Request对象分别打上对应Tab的tag。并在滑动时取消这些请求，用来确保新生成的tab不会被前面tab的请求任务所卡到。

下面一个使用String来打Tag的例子：

1. 定义你的tag并添加到你的请求任务中。

```
public static final String TAG = "MyTag";
StringRequest stringRequest; // Assume this exists.
RequestQueue mRequestQueue;  // Assume this exists.

// Set the tag on the request.
stringRequest.setTag(TAG);

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
```

1. 在activity的onStop()方法里面，取消所有的包含这个tag的请求任务。

```
@Override
protected void onStop () {
    super.onStop();
    if (mRequestQueue != null) {
        mRequestQueue.cancelAll(TAG);
    }
}
```

当取消请求时请注意：如果你依赖你的响应handler来标记状态或者触发另外一个进程，你需要对此进行考虑。再说一次，response handler是不会被执行的。

建立请求队列（RequestQueue）

> 编写:kesenhoo - 原文:http://developer.android.com/training/volley/requestqueue.html

前一节课演示了如何使用 `Volley.newRequestQueue` 这一简便的方法来建立一个 `RequestQueue`，这是利用了 Volley 默认行为的优势。这节课会介绍如何显式地建立一个 `RequestQueue`，以便满足我们自定义的需求。

这节课同样会介绍一种推荐的实现方式：创建一个单例的 `RequestQueue`，这使得 `RequestQueue` 能够持续保持在我们 app 的生命周期中。

建立网络和缓存

一个 `RequestQueue` 需要两部分来支持它的工作：一部分是网络操作，用来传输请求，另外一个是用来处理缓存操作的 Cache。在 Volley 的工具箱中包含了标准的实现方式：`DiskBasedCache` 提供了每个文件与对应响应数据一一映射的缓存实现。 `BasicNetwork` 提供了一个基于 AndroidHttpClient 或者 HttpURLConnection 的网络传输。

`BasicNetwork` 是 Volley 默认的网络操作实现方式。一个 `BasicNetwork` 必须使用我们的 app 用于连接网络的 HTTP Client 进行初始化。这个 Client 通常是AndroidHttpClient 或者 HttpURLConnection：

- 对于 app target API level 低于 API 9（Gingerbread）的使用 AndroidHttpClient。在 Gingerbread 之前，HttpURLConnection 是不可靠的。对于这个的细节，请参考 Android's HTTP Clients。
- 对于 API Level 9 以及以上的，使用 HttpURLConnection。

我们可以通过检查系统版本选择合适的 HTTP Client，从而创建一个能够运行在所有 Android 版本上的应用。例如：

```
HttpStack stack;
...
// If the device is running a version >= Gingerbread...
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
    // ...use HttpURLConnection for stack.
} else {
    // ...use AndroidHttpClient for stack.
}
Network network = new BasicNetwork(stack);
```

下面的代码片段演示了如何一步步建立一个 `RequestQueue`：

```
RequestQueue mRequestQueue;

// Instantiate the cache
Cache cache = new DiskBasedCache(getCacheDir(), 1024 * 1024); // 1MB cap

// Set up the network to use HttpURLConnection as the HTTP client.
Network network = new BasicNetwork(new HurlStack());

// Instantiate the RequestQueue with the cache and network.
mRequestQueue = new RequestQueue(cache, network);

// Start the queue
mRequestQueue.start();

String url ="http://www.myurl.com";

// Formulate the request and handle the response.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
        new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Do something with the response
    }
},
    new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            // Handle error
    }
});

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
...
```

如果我们仅仅是想做一个单次的请求并且不想要线程池一直保留，我们可以通过使用在前面一课：发送一个简单的请求（Sending a Simple Request）文章中提到的 `Volley.newRequestQueue()` 方法，在任何需要的时刻创建 `RequestQueue`，然后在我们的响应回调里面执行 `stop()` 方法来停止操作。但是更通常的做法是创建一个 `RequestQueue` 并设置为一个单例。下面部分将演示这种做法。

使用单例模式

如果我们的应用需要持续地使用网络，更加高效的方式应该是建立一个 `RequestQueue` 的单例，这样它能够持续保持在整个 app 的生命周期中。我们可以通过多种方式来实现这个单例。推荐的方式是实现一个单例类，里面封装了 `RequestQueue` 对象与其它的 Volley 功能。

另外一个方法是继承 `Application` 类，并在 `Application.OnCreate()` 方法里面建立 `RequestQueue` 。但是我们并不推荐这个方法，因为一个 static 的单例能够以一种更加模块化的方式提供同样的功能。

一个关键的概念是 `RequestQueue` 必须使用 Application context 来实例化，而不是 Activity context。这确保了 `RequestQueue` 在我们 app 的生命周期中一直存活，而不会因为 activity 的重新创建而被重新创建(例如，当用户旋转设备时)。

下面是一个单例类，提供了 `RequestQueue` 与 `ImageLoader` 功能：

```
public class MySingleton {
    private static MySingleton mInstance;
    private RequestQueue mRequestQueue;
    private ImageLoader mImageLoader;
    private static Context mCtx;

    private MySingleton(Context context) {
        mCtx = context;
        mRequestQueue = getRequestQueue();

        mImageLoader = new ImageLoader(mRequestQueue,
                new ImageLoader.ImageCache() {
            private final LruCache<String, Bitmap>
                    cache = new LruCache<String, Bitmap>(20);

            @Override
            public Bitmap getBitmap(String url) {
                return cache.get(url);
            }

            @Override
            public void putBitmap(String url, Bitmap bitmap) {
                cache.put(url, bitmap);
            }
        });
    }

    public static synchronized MySingleton getInstance(Context context) {
        if (mInstance == null) {
            mInstance = new MySingleton(context);
        }
        return mInstance;
    }

    public RequestQueue getRequestQueue() {
        if (mRequestQueue == null) {
            // getApplicationContext() is key, it keeps you from leaking the
            // Activity or BroadcastReceiver if someone passes one in.
            mRequestQueue = Volley.newRequestQueue(mCtx.getApplicationContext());
        }
        return mRequestQueue;
    }

    public <T> void addToRequestQueue(Request<T> req) {
        getRequestQueue().add(req);
    }

    public ImageLoader getImageLoader() {
        return mImageLoader;
    }
}
```

下面演示了利用单例类来执行 `RequestQueue` 的操作：

```
// Get a RequestQueue
RequestQueue queue = MySingleton.getInstance(this.getApplicationContext()).
    getRequestQueue();
...

// Add a request (in this example, called stringRequest) to your RequestQueue.
MySingleton.getInstance(this).addToRequestQueue(stringRequest);
```

创建标准的网络请求

> 编写:[kesenhoo](https://github.com/kesenhoo) - 原文:http://developer.android.com/training/volley/request.html

这一课会介绍如何使用 Volley 支持的常用请求类型：

- `StringRequest`。指定一个 URL 并在响应回调中接收一个原始的字符串数据。请参考前一课的示例。
- `ImageRequest`。指定一个 URL 并在响应回调中接收一个图片。
- `JsonObjectRequest` 与 `JsonArrayRequest` （均为 `JsonRequest` 的子类）。指定一个 URL 并在响应回调中获取到一个 JSON 对象或者 JSON 数组。

如果我们需要的是上面演示的请求类型，那么我们很可能不需要实现一个自定义的请求。这节课会演示如何使用那些标准的请求类型。关于如何实现自定义的请求，请看下一课：实现自定义的请求。

请求一张图片

Volley 为请求图片提供了如下的类。这些类依次有着依赖关系，用来支持在不同的层级进行图片处理：

- `ImageRequest` —— 一个封装好的，用来处理 URL 请求图片并且返回一张解完码的位图（bitmap）。它同样提供了一些简便的接口方法，例如指定一个大小进行重新裁剪。它的主要好处是 Volley 会确保类似 decode，resize 等耗时的操作在工作线程中执行。

- `ImageLoader` —— 一个用来处理加载与缓存从网络上获取到的图片的帮助类。`ImageLoader` 是大量 `ImageRequest` 的协调器。例如，在 `ListView` 中需要显示大量缩略图的时候。`ImageLoader` 为通常的 Volley cache 提供了更加前瞻的内存缓存，这个缓存对于防止图片抖动非常有用。这还使得在不阻塞或者延迟主线程的前提下实现缓存命中（这对于使用磁盘 I/O 是无法实现的）。`ImageLoader` 还能够实现响应联合（response coalescing），避免几乎每一个响应回调里面都设置 bitmap 到 view 上面。响应联合使得能够同时提交多个响应，这提升了性能。

- `NetworkImageView` —— 在 `ImageLoader` 的基础上建立，并且在通过网络 URL 取回的图片的情况下，有效地替换 `ImageView`。如果 view 从层次结构中分离，`NetworkImageView` 也可以管理取消挂起请求。

使用 ImageRequest

下面是一个使用 `ImageRequest` 的示例。它会获取 URL 上指定的图片并显示到 app 上。注意到，里面演示的 `RequestQueue` 是通过上一课提到的单例类实现的：

```
ImageView mImageView;
String url = "http://i.imgur.com/7spzG.png";
mImageView = (ImageView) findViewById(R.id.myImage);
...

// Retrieves an image specified by the URL, displays it in the UI.
ImageRequest request = new ImageRequest(url,
    new Response.Listener() {
        @Override
        public void onResponse(Bitmap bitmap) {
            mImageView.setImageBitmap(bitmap);
        }
    }, 0, 0, null,
    new Response.ErrorListener() {
        public void onErrorResponse(VolleyError error) {
            mImageView.setImageResource(R.drawable.image_load_error);
        }
    });
// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(request);
```

使用 ImageLoader 和 NetworkImageView

我们可以使用 `ImageLoader` 与 `NetworkImageView` 来有效地管理类似 ListView 等显示多张图片的情况。在 layout XML 文件中，我们以与使用 ImageView 差不多的方法使用 `NetworkImageView` ，例如:

```
<com.android.volley.toolbox.NetworkImageView
        android:id="@+id/networkImageView"
        android:layout_width="150dp"
        android:layout_height="170dp"
        android:layout_centerHorizontal="true" />
```

我们可以使用 `ImageLoader` 自身来显示一张图片，例如：

```
ImageLoader mImageLoader;
ImageView mImageView;
// The URL for the image that is being loaded.
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...
mImageView = (ImageView) findViewById(R.id.regularImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();
mImageLoader.get(IMAGE_URL, ImageLoader.getImageListener(mImageView,
         R.drawable.def_image, R.drawable.err_image));
```

然而，如果我们要做的是为 `ImageView` 进行图片设置，那么我们可以使用 `NetworkImageView` 来实现，例如：

```
ImageLoader mImageLoader;
NetworkImageView mNetworkImageView;
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...

// Get the NetworkImageView that will display the image.
mNetworkImageView = (NetworkImageView) findViewById(R.id.networkImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();

// Set the URL of the image that should be loaded into this view, and
// specify the ImageLoader that will be used to make the request.
mNetworkImageView.setImageUrl(IMAGE_URL, mImageLoader);
```

上面的代码是通过通过前一节课讲到的单例类来访问 `RequestQueue` 与 `ImageLoader` 。这种方法保证了我们的 app 创建这些类的单例会持续存在于 app 的生命周期。这对于 `ImageLoader` （一个用来处理加载与缓存图片的帮助类）很重要的原因是：内存缓存的主要功能是允许非抖动旋转。使用单例模式可以使得 bitmap 的缓存比 activity 存在的时间长。如果我们在 activity 中创建 `ImageLoader` ，这个 `ImageLoader` 有可能会在每次旋转设备的时候都被重新创建。这可能会导致抖动。

举一个 LRU cache 的例子

Volley 工具箱中提供了一种通过 `DiskBasedCache` 类实现的标准缓存。这个类能够缓存文件到磁盘的指定目录。但是为了使用 `ImageLoader` ，我们应该提供一个自定义的内存 LRC bitmap 缓存，这个缓存实现了 `ImageLoader.ImageCache` 接口。我们可能想把缓存设置成一个单例。关于更多的有关内容，请参考建立请求队列.

下面是一个内存 `LruBitmapCache` 类的实现示例。它继承 LruCache 类并实现了 `ImageLoader.ImageCache` 接口：

```
import android.graphics.Bitmap;
import android.support.v4.util.LruCache;
import android.util.DisplayMetrics;
import com.android.volley.toolbox.ImageLoader.ImageCache;

public class LruBitmapCache extends LruCache<String, Bitmap>
        implements ImageCache {

    public LruBitmapCache(int maxSize) {
        super(maxSize);
    }

    public LruBitmapCache(Context ctx) {
        this(getCacheSize(ctx));
    }

    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getRowBytes() * value.getHeight();
    }

    @Override
    public Bitmap getBitmap(String url) {
        return get(url);
    }

    @Override
    public void putBitmap(String url, Bitmap bitmap) {
        put(url, bitmap);
    }

    // Returns a cache size equal to approximately three screens worth of images.
    public static int getCacheSize(Context ctx) {
        final DisplayMetrics displayMetrics = ctx.getResources().
                getDisplayMetrics();
        final int screenWidth = displayMetrics.widthPixels;
        final int screenHeight = displayMetrics.heightPixels;
        // 4 bytes per pixel
        final int screenBytes = screenWidth * screenHeight * 4;

        return screenBytes * 3;
    }
}
```

下面是如何实例化一个 `ImageLoader` 来使用这个 cache:

```
RequestQueue mRequestQueue; // assume this exists.
ImageLoader mImageLoader = new ImageLoader(mRequestQueue, new LruBitmapCache(LruBitmap
Cache.getCacheSize()));
```

请求 JSON

Volley 提供了以下的类用来执行 JSON 请求：

- `JsonArrayRequest` —— 一个为了获取给定 URL 的 JSONArray 响应正文的请求。
- `JsonObjectRequest` —— 一个为了获取给定 URL 的 JSONObject 响应正文的请求。允许传进一个可选的 JSONObject 作为请求正文的一部分。

这两个类都是基于一个公共基类 `JsonRequest` 。我们遵循我们在其它请求类型使用的同样的基本模式来使用这些类。如下演示了如何获取一个 JSON feed 并显示到 UI 上：

```
TextView mTxtDisplay;
ImageView mImageView;
mTxtDisplay = (TextView) findViewById(R.id.txtDisplay);
String url = "http://my-json-feed";

JsonObjectRequest jsObjRequest = new JsonObjectRequest
        (Request.Method.GET, url, null, new Response.Listener() {

    @Override
    public void onResponse(JSONObject response) {
        mTxtDisplay.setText("Response: " + response.toString());
    }
}, new Response.ErrorListener() {

    @Override
    public void onErrorResponse(VolleyError error) {
        // TODO Auto-generated method stub

    }
});

// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(jsObjRequest);
```

关于基于 Gson 实现一个自定义的 JSON 请求对象，请参考下一节课：实现一个自定义的请求。

实现自定义的网络请求

> 编写:kesenhoo - 原文:http://developer.android.com/training/volley/request-custom.html

这节课会介绍如何实现自定义的请求类型，这些自定义的类型不属于 Volley 内置支持包里面。

编写一个自定义请求

大多数的请求类型都已经包含在 Volley 的工具箱里面。如果我们的请求返回数值是一个 string，image 或者 JSON，那么是不需要自己去实现请求类的。

对于那些需要自定义的请求类型，我们需要执行以下操作：

- 继承 `Request<T>` 类， `<T>` 表示解析过的响应请求预期的数据类型。因此如果我们需要解析的响应类型是一个 String，可以通过继承 `Request<String>` 来创建自定义的请求。请参考 Volley 工具类中的 `StringRequest` 与 `ImageRequest` 来学习如何继承 `Request<T>` 。
- 实现抽象方法 `parseNetworkResponse()` 与 `deliverResponse()` ，下面会详细介绍。

parseNetworkResponse

一个 `Response` 封装了用于发送的给定类型（例如，string、image、JSON等）解析过的响应。下面会演示如何实现 `parseNetworkResponse()` ：

```
@Override
protected Response<T> parseNetworkResponse(
        NetworkResponse response) {
    try {
        String json = new String(response.data,
        HttpHeaderParser.parseCharset(response.headers));
    return Response.success(gson.fromJson(json, clazz),
    HttpHeaderParser.parseCacheHeaders(response));
    }
    // handle errors
...
}
```

请注意：

- `parseNetworkResponse()` 的参数是类型是 `NetworkResponse` ，这种参数以 byte[]、HTTP status code 以及 response headers 的形式包含响应负载。

- 我们实现的方法必须返回一个 `Response<T>` ，它包含了我们指定类型的响应对象与缓存 metadata 或者是一个错误。

如果我们的协议没有标准的缓存机制，那么我们可以自己建立一个 `Cache.Entry` , 但是大多数请求都可以用下面的方式来处理:

```
return Response.success(myDecodedObject,
        HttpHeaderParser.parseCacheHeaders(response));
```

Volley 在工作线程中执行 `parseNetworkResponse()` 方法。这确保了耗时的解析操作，例如 decode 一张 JPEG 图片成 bitmap，不会阻塞 UI 线程。

deliverResponse

Volley 会把 `parseNetworkResponse()` 方法返回的数据带到主线程的回调中。如下所示：

```
protected void deliverResponse(T response) {
        listener.onResponse(response);
```

Example: GsonRequest

Gson 是一个使用映射支持 JSON 与 Java 对象之间相互转换的库文件。我们可以定义与 JSON keys 相对应名称的 Java 对象。把对象传递给 Gson，然后 Gson 会帮我们为对象填充字段值。下面是一个完整的示例：演示了使用 Gson 解析 Volley 数据：

```
public class GsonRequest<T> extends Request<T> {
    private final Gson gson = new Gson();
    private final Class<T> clazz;
    private final Map<String, String> headers;
    private final Listener<T> listener;

    /**
     * Make a GET request and return a parsed object from JSON.
     *
     * @param url URL of the request to make
     * @param clazz Relevant class object, for Gson's reflection
     * @param headers Map of request headers
     */
    public GsonRequest(String url, Class<T> clazz, Map<String, String> headers,
            Listener<T> listener, ErrorListener errorListener) {
        super(Method.GET, url, errorListener);
        this.clazz = clazz;
        this.headers = headers;
        this.listener = listener;
    }

    @Override
    public Map<String, String> getHeaders() throws AuthFailureError {
        return headers != null ? headers : super.getHeaders();
    }

    @Override
    protected void deliverResponse(T response) {
        listener.onResponse(response);
    }

    @Override
    protected Response<T> parseNetworkResponse(NetworkResponse response) {
        try {
            String json = new String(
                    response.data,
                    HttpHeaderParser.parseCharset(response.headers));
            return Response.success(
                    gson.fromJson(json, clazz),
                    HttpHeaderParser.parseCacheHeaders(response));
        } catch (UnsupportedEncodingException e) {
            return Response.error(new ParseError(e));
        } catch (JsonSyntaxException e) {
            return Response.error(new ParseError(e));
        }
    }
}
```

如果你愿意使用的话，Volley 提供了现成的 `JsonArrayRequest` 与 `JsonArrayObject` 类。参考上一课创建标准的网络请求。

Android联系人信息与位置信息

> 编写:spencer198711,Muyangmin - 原文:http://developer.android.com/training/building-userinfo.html

这几节课为大家介绍如何在我们的app中添加用户个人信息。我们可以通过识别用户，提供用户相关信息和提供用户周围的位置信息等方法来添加个人信息。

Lessons

访问联系人数据 - Accessing Contacts Data

如何使用Android的Contacts Provider来显示和修改联系人信息。

位置信息- Making Your App Location-Aware

如何通过获得用户当前位置来给我们的App添加定位功能(位置感知)。

联系人信息

> 编写:spencer198711 - 原文:http://developer.android.com/training/contacts-provider/index.html

Contacts Provider是用户联系人信息的集中仓库， 它包含了来自联系人应用与社交应用的联系人数据。在我们的应用中，我们可以通过调用**ContentResolver**方法或者通过发送Intent给联系人应用来访问Contacts Provider的信息。

这个章节会讲解获取联系人列表，显示指定联系人详情以及通过intent来修改联系人信息。这里介绍的基础技能能够扩展到执行更复杂的任务。另外，这个章节也会帮助我们了解Contacts Provider的整个架构与操作方法。

Lessons

获取联系人列表

学习如何获取联系人列表。你可以使用下面的技术来筛选需要的信息：

- 通过联系人名字进行筛选
- 通过联系人类型进行筛选
- 通过类似电话号码等指定的一类信息进行筛选。

获取联系人详情

学习如何获取单个联系人的详情。一个联系人的详细信息包括电话号码与邮件地址等等。你可以获取所有的详细信息，也有可以只获取指定类型的详细数据，例如邮件地址。

使用**Intents**修改联系人信息

学习如何通过发送intent给联系人应用来修改联系人信息。

显示联系人头像

学习如何显示**QuickContactBadge**小组件。当用户点击联系人臂章(头像)组件时，会打开一个对话框，这个对话框会显示联系人详情，并提供操作按钮来处理详细信息。例如，如果联系人信息有邮件地址，这个对话框可以显示一个启动默认邮件应用的操作按钮。

获取联系人列表

> 编写:spencer198711 - 原文:http://developer.android.com/training/contacts-provider/retrieve-names.html

这一课展示了如何根据要搜索的字符串去匹配联系人的数据，从而得到联系人列表，你可以使用以下方法去实现：

匹配联系人名字

通过搜索字符串来匹配联系人名字的全部或者部分来获得联系人列表。因为Contacts Provider允许多个实例拥有相同的名字，所以这种方法能够返回匹配的列表。

匹配特定的数据类型，比如电话号码

通过搜索字符串来匹配联系人的某一特定数据类型（如电子邮件地址），来取得符合要求的联系人列表。例如，这种方法可以列出电子邮件地址与搜索字符相匹配的所有联系人。

匹配任意类型的数据

通过搜索字符串来匹配联系人详情的所有数据类型，包括名字、电话号码、地址、电子邮件地址等等。例如，这种方法接受任意数据类型的搜索字符串，并列出与这个搜索字符串相匹配的联系人。

> **Note**：这一课的所有例子都使用CursorLoader获取Contacts Provider中的数据。CursorLoader在一个与UI线程相独立的工作线程进行查询操作。这保证了数据查询不会降低UI响应的时间，以免引起槽糕的用户体验。更多信息，请参照在后台加载数据。

请求读取联系人的权限

为了能够在Contacts Provider中做任意类型的搜索，我们的应用必须拥有READ_CONTACTS权限。为了拥有这个权限，我们需要在项目的manifest文件的节点中添加子结点，如下：

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

根据名字取得联系人并列出结果

这种方法根据搜索字符串，去匹配Contacts Provider的ContactsContract.Contacts表中的联系人名字。通常希望在ListView中展示结果，去让用户在所有匹配的联系人中做选择。

定义ListView和列表项的布局

为了能够将搜索结果展示在列表中，我们需要一个包含ListView以及其他布局控件的主布局文件，和一个定义列表中每一项的布局文件。例如，可以使用以下XML代码去创建主布局文件res/layout/contacts_list_view.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@android:id/list"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
```

这个XML代码使用了Android内建的ListView控件,他的id是android:id/list。

使用以下XML代码定义列表项布局文件contacts_list_item.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@android:id/text1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:clickable="true"/>
```

这个XML代码使用了Android内建的TextView控件,他的id是android:text1。

> **Note**：本课并不会描述如何从用户那里获取搜索字符串的界面，因为我们可能会间接地获取这个字符串。比如说，我们可能会给用户一个选项去输入文字信息，把这些文字信息作为搜索字符串去匹配联系人的名字。

刚刚写的这两个布局文件定义了一个显示ListView的用户界面。下一步是编写使用这个用户界面显示联系人列表的代码。

定义一个显示联系人列表的Fragment

为了显示联系人列表，需要定义一个由Activity加载的Fragment。使用Fragment是一个比较灵活的方法，因为我们可以使用一个Fragment去显示列表，用另一个Fragment显示用户在列表中选择的联系人的详情。使用这种方式，我们可以将本课程中展示的方法和另外一课获取联系人详情的方法联系起来。

想要学习如何在Activity中使用一个或者多个Fragment，请阅读培训课程使用Fragment建立动态UI。

为了方便我们编写对Contacts Provider的查询，Android框架提供了一个叫做ContactsContract的契约类，这个类定义了一些对查询Contacts Provider很有用的常量和方法。当我们使用这个类的时候，我们不用自己定义内容URI、表名、列名等常量。使用这个

类，只需要引入以下类声明：

```
import android.provider.ContactsContract;
```

由于代码中使用了CursorLoader去获取provider的数据，所以我们必须实现加载器接口LoaderManager.LoaderCallbacks。同时，为了检测用户从结果列表中选择了哪一个联系人，必须实现适配器接口AdapterView.OnItemClickListener。例如：

```
...
import android.support.v4.app.Fragment;
import android.support.v4.app.LoaderManager.LoaderCallbacks;
import android.widget.AdapterView;
...
public class ContactsFragment extends Fragment implements
    LoaderManager.LoaderCallbacks<Cursor>,
    AdapterView.OnItemClickListener {
```

定义全局变量

定义在其他代部分码中使用的全局变量：

```
...
/*
 * Defines an array that contains column names to move from
 * the Cursor to the ListView.
 */
@SuppressLint("InlinedApi")
private final static String[] FROM_COLUMNS = {
        Build.VERSION.SDK_INT
                >= Build.VERSION_CODES.HONEYCOMB ?
                Contacts.DISPLAY_NAME_PRIMARY :
                Contacts.DISPLAY_NAME
};
/*
 * Defines an array that contains resource ids for the layout views
 * that get the Cursor column contents. The id is pre-defined in
 * the Android framework, so it is prefaced with "android.R.id"
 */
private final static int[] TO_IDS = {
       android.R.id.text1
};
// Define global mutable variables
// Define a ListView object
ListView mContactsList;
// Define variables for the contact the user selects
// The contact's _ID value
long mContactId;
// The contact's LOOKUP_KEY
String mContactKey;
// A content URI for the selected contact
Uri mContactUri;
// An adapter that binds the result Cursor to the ListView
private SimpleCursorAdapter mCursorAdapter;
...
```

> **Note**：由于Contacts.DISPLAY_NAME_PRIMARY需要在Android 3.0（API版本11）或者更高的版本才能使用，如果应用的minSdkVersion是10或者更低，会在eclipse中产生警告信息。为了关闭这个警告，我们可以在FROM_COLUMNS定义之前加上@SuppressLint("InlinedApi")注解。

初始化Fragment

为了初始化Fragment，Android系统需要我们为这个Fragment添加空的、公有的构造方法，同时在回调方法onCreateView()中绑定界面。例如：

```
// Empty public constructor, required by the system
public ContactsFragment() {}
// A UI Fragment must inflate its View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    // Inflate the fragment layout
    return inflater.inflate(R.layout.contact_list_fragment,
        container, false);
}
```

为ListView设置CursorAdapter

设置SimpleCursorAdapter，将搜索结果绑定到ListView。为了获得显示联系人列表的ListView控件，需要使用Fragment的父Activity调用Activity.findViewById()。当调用setAdapter()的时候，需要使用父Activity的上下文（Context）。

```
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    ...
    // Gets the ListView from the View list of the parent activity
    mContactsList =
        (ListView) getActivity().findViewById(R.layout.contact_list_view);
    // Gets a CursorAdapter
    mCursorAdapter = new SimpleCursorAdapter(
            getActivity(),
            R.layout.contact_list_item,
            null,
            FROM_COLUMNS, TO_IDS,
            0);
    // Sets the adapter for the ListView
    mContactsList.setAdapter(mCursorAdapter);
}
```

为选择的联系人设置监听器

当我们显示搜索列表结果的时候，我们通常会让用户选择某一个联系人去做进一步的处理。例如，当用户选择某一个联系人的时候，可以在地图上显示这个联系人的地址。为了能够提供这个功能，我们需要定义当前的Fragment为一个点击监听器，这需要这个类实现AdapterView.OnItemClickListener接口，就像前面介绍的定义显示联系人列表的**Fragment**那样。

继续设置这个监听器，需要在onActivityCreated()方法中调用setOnItemClickListener()以使得这个监听器绑定到ListView。例如：

```
public void onActivityCreated(Bundle savedInstanceState) {
    ...
    // Set the item click listener to be the current fragment.
    mContactsList.setOnItemClickListener(this);
    ...
}
```

由于指定了当前的Fragment作为ListView的点击监听器，现在我们需要实现处理点击事件的onItemClick()方法。这个会在随后讨论。

定义查询映射

定义一个常量，这个常量包含我们想要从查询结果中返回的列。Listview中的每一项显示了一个联系人的名字。在Android 3.0（API version 11）或者更高的版本，这个列的名字是Contacts.DISPLAY_NAME_PRIMARY；在Android 3.0之前，这个列的名字是Contacts.DISPLAY_NAME。

在SimpleCursorAdapter绑定过程中会用到Contacts._ID列。 Contacts._ID和LOOKUP_KEY一同用来构建用户选择的联系人的内容URI。

```
...
@SuppressLint("InlinedApi")
private static final String[] PROJECTION = {
    Contacts._ID,
    Contacts.LOOKUP_KEY,
    Build.VERSION.SDK_INT
            >= Build.VERSION_CODES.HONEYCOMB ?
            Contacts.DISPLAY_NAME_PRIMARY :
            Contacts.DISPLAY_NAME
};
```

定义Cursor的列索引常量

为了从Cursor中获得单独某一列的数据，我们需要知道这一列在Cursor中的索引值。我们需要定义Cursor列的索引值，这些索引值与我们定义查询映射的列的顺序是一样的。例如：

```
// The column index for the _ID column
private static final int CONTACT_ID_INDEX = 0;
// The column index for the LOOKUP_KEY column
private static final int LOOKUP_KEY_INDEX = 1;
```

指定查询标准

为了指定我们想要的数据，我们需要创建一个包含文本表达式和变量的组合，去告诉provider我们需要的数据列和想要的值。

对于文本表达式，定义一个常量，列出所有搜索到的列。尽管这个表达式可以包含变量值，但是建议用"?"占位符来替代这个值。在搜索的时候，占位符里的值会被数组里的值所取代。使用"?"占位符确保了搜索条件是由绑定产生而不是由SQL编译产生。这个方法消除了恶意SQL注入的可能。例如：

```
// Defines the text expression
@SuppressLint("InlinedApi")
private static final String SELECTION =
        Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
        Contacts.DISPLAY_NAME_PRIMARY + " LIKE ?" :
        Contacts.DISPLAY_NAME + " LIKE ?";
// Defines a variable for the search string
private String mSearchString;
// Defines the array to hold values that replace the ?
private String[] mSelectionArgs = { mSearchString };
```

定义onItemClick()方法

在之前的内容中，我们为Listview设置了列表项点击监听器，现在需要定义AdapterView.OnItemClickListener.onItemClick()方法以实现监听器行为：

```
@Override
public void onItemClick(
    AdapterView<?> parent, View item, int position, long rowID) {
    // Get the Cursor
    Cursor cursor = parent.getAdapter().getCursor();
    // Move to the selected contact
    cursor.moveToPosition(position);
    // Get the _ID value
    mContactId = getLong(CONTACT_ID_INDEX);
    // Get the selected LOOKUP KEY
    mContactKey = getString(CONTACT_KEY_INDEX);
    // Create the contact's content Uri
    mContactUri = Contacts.getLookupUri(mContactId, mContactKey);
    /*
     * You can use mContactUri as the content URI for retrieving
     * the details for a contact.
     */
}
```

初始化Loader

由于使用了CursorLoader获取数据，我们必须初始化后台线程和其他的控制异步获取数据的变量。需要在onActivityCreated()方法中做初始化的工作，这个方法是在Fragment的界面显示之前被调用的，相关代码展示如下：

```
public class ContactsFragment extends Fragment implements
    LoaderManager.LoaderCallbacks<Cursor> {
...
    // Called just before the Fragment displays its UI
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        // Always call the super method first
        super.onActivityCreated(savedInstanceState);
        ...
        // Initializes the loader
        getLoaderManager().initLoader(0, null, this);
```

实现onCreateLoader()方法

我们需要实现onCreateLoader()方法，这个方法是在调用initLoader()后马上被loader框架调用的。

在onCreateLoader()方法中，设置搜索字符串模式。为了让一个字符串符合一个模式，插入"%"字符代表0个或多个字符，插入"_"代表一个字符。例如，模式%Jefferson%将会匹配“Thomas Jefferson”和“Jefferson Davis”。

这个方法返回一个CursorLoader对象。对于内容URI，则使用了Contacts.CONTENT_URI，这个URI关联到整个表，例子如下所示：

```
...
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    /*
     * Makes search string into pattern and
     * stores it in the selection array
     */
    mSelectionArgs[0] = "%" + mSearchString + "%";
    // Starts the query
    return new CursorLoader(
            getActivity(),
            Contacts.CONTENT_URI,
            PROJECTION,
            SELECTION,
            mSelectionArgs,
            null
    );
}
```

实现onLoadFinished()方法和onLoaderReset()方法

实现onLoadFinished()方法。当Contacts Provider返回查询结果的时候，loader框架会调用onLoadFinished()方法。在这个方法中，将查询结果Cursor传给SimpleCursorAdapter，这将会使用这个搜索结果自动更新ListView。

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    // Put the result Cursor in the adapter for the ListView
    mCursorAdapter.swapCursor(cursor);
}
```

当loader框架检测到结果集Cursor包含过时的数据时，它会调用onLoaderReset()。我们需要删除SimpleCursorAdapter对已经存在Cursor的引用。如果不这么做的话，loader框架将不会回收Cursor对象，这将会导致内存泄漏。例如：

```
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    // Delete the reference to the existing Cursor
    mCursorAdapter.swapCursor(null);
}
```

我们现在已经实现了一个应用的关键部分，即根据搜索字符串匹配联系人名字和将获得的结果展示在ListView中。用户可以点击选择一个联系人名字，这将会触发一个监听器，在监听器的回调函数中，你可以使用此联系人的数据做进一步的处理。例如，你可以进一步获取此联系人的详情，想要知道何如获取联系人详情，请继续学习下一课获取联系人详情。

想要了解更多搜索用户界面的知识，请参考API指南Creating a Search Interface。

这一课的以下内容展示了在Contacts Provider中查找联系人的其他方法。

根据特定的数据类型匹配联系人

这种方法可以让我们指定想要匹配的数据类型。根据名字去检索是这种类型的查询的一个具体例子。但也可以用任何与联系人详情数据相关的数据类型去做查询。例如，我们可以检索具有特定邮政编码联系人，在这种情况下，搜索字符串将会去匹配存储在一个邮政编码列中的数据。

为了实现这种类型的检索，首先实现以下的代码，正如之前的内容所展示的：

- 请求读取联系人的权限
- 定义列表和列表项的布局
- 定义显示联系人列表的Fragment
- 定义全局变量

- 初始化Fragment
- 为ListView设置CursorAdapter
- 设置选择联系人的监听器
- 定义Cursor的列索引常量

  尽管我们现在从不同的表中取数据，检索列的映射顺序是一样的，所以我们可以为这个Cursor使用同样的索引常量。

- 定义onItemClick()方法
- 初始化loader
- 实现onLoadFinished()方法和onLoaderReset()方法

为了将搜索字符串匹配特定的详请数据类型并显示结果，以下的步骤展示了我们需要额外添加的代码。

选择要查询的数据类型和数据库表

为了从特定类型的详请数据中查询，我们必须知道的数据类型的自定义MIME类型的值。每一个数据类型拥有唯一的`MIME`类型值，这个值在ContactsContract.CommonDataKinds的子类中被定义为常量`CONTENT_ITEM_TYPE`，并且与实际的数据类型相关。子类的名字会表明它们的实际数据类型。例如，email数据的子类是`ContactsContract.CommonDataKinds.Email`，并且email的自定义MIME类型是`Email.CONTENT_ITEM_TYPE`。

在搜索中需要使用ContactsContract.Data类。同时所有需要的常量，包括数据映射、选择字句、排序规则都是由这个类定义或继承自此类。

定义查询映射

为了定义一个查询映射，请选择一个或者多个定义在ContactsContract.Data表或其子类的列。Contacts Provider在返回行结果集之前，隐式的连接了ContactsContract.Data表和其他表。例如：

```
@SuppressLint("InlinedApi")
private static final String[] PROJECTION = {
    /*
     * The detail data row ID. To make a ListView work,
     * this column is required.
     */
    Data._ID,
    // The primary display name
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
            Data.DISPLAY_NAME_PRIMARY :
            Data.DISPLAY_NAME,
    // The contact's _ID, to construct a content URI
    Data.CONTACT_ID
    // The contact's LOOKUP_KEY, to construct a content URI
    Data.LOOKUP_KEY (a permanent link to the contact
};
```

定义查询标准

为了根据特定的联系人数据类型查询字符串，请按照以下方法构建查询选择子句：

- 包含搜索字符串的列名。这个名字根据数据类型所变化，所以我们需要找到与数据类型对应的ContactsContract.CommonDataKinds的子类，并从这个子类中选择列名。例如，想要搜索email地址，需要使用Email.ADDRESS列。
- 搜索字符串本身，请在查询选择子句里使用"?"表示。
- 列名包含自定义的MIME类型值。这个列名字总是Data.MIMETYPE。
- 自定义MIME类型值的数据类型。如之前描述，这需要使用ContactsContract.CommonDataKinds子类中的 `CONTENT_ITEM_TYPE` 常量。例如，email数据的MIME类型值是 `Email.CONTENT_ITEM_TYPE` 。需要在这个常量值的开头和结尾加上"'"（单引号）。否则，provider会把这个值翻译成一个变量而不是一个字符串。我们不需要为这个值提供占位符，因为我们在使用一个常量而不是用户提供的值。例如：

```
/*
 * Constructs search criteria from the search string
 * and email MIME type
 */
private static final String SELECTION =
    /*
     * Searches for an email address
     * that matches the search string
     */
    Email.ADDRESS + " LIKE ? " + "AND " +
    /*
     * Searches for a MIME type that matches
     * the value of the constant
     * Email.CONTENT_ITEM_TYPE. Note the
     * single quotes surrounding Email.CONTENT_ITEM_TYPE.
     */
    Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'";
```

下一步，定义包含选择字符串的变量：

```
String mSearchString;
String[] mSelectionArgs = { "" };
```

实现onCreateLoader()方法

现在，我们已经详述了想要的数据和如何找到这些数据，如何在onCreateLoader()方法中定义一个查询。使用你的数据映射、查询选择表达式和一个数组作为选择表达式的参数，并从这个方法中返回一个新的CursorLoader对象。而内容URI需要使用Data.CONTENT_URI，例如：

```
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    // OPTIONAL: Makes search string into pattern
    mSearchString = "%" + mSearchString + "%";
    // Puts the search string into the selection criteria
    mSelectionArgs[0] = mSearchString;
    // Starts the query
    return new CursorLoader(
            getActivity(),
            Data.CONTENT_URI,
            PROJECTION,
            SELECTION,
            mSelectionArgs,
            null
    );
}
```

这段代码片段是基于特定的联系人详情数据类型的简单反向查找。如果我们的应用关注于某一种特定的数据类型，比如说email地址，并且允许用户获得与此数据相关的联系人名字，这种形式的查询是最好的方法。

根据任意类型的数据匹配联系人

根据任意数据类型获取联系人时，如果联系人的数据（这些数据包括名字、email地址、邮件地址和电话号码等等）能匹配要搜索的字符串，那么该联系人信息将会被返回。这种搜索结果会比较广泛。例如，如果搜索字符串是"Doe"，搜索任意类型的数据将会返回名字为"Jone Doe"的联系人，也会返回一个住在"Doe Street"的联系人。

为了完成这种类型的查询，就像之前展示的那样，首先需要实现以下代码：

- 请求读取联系人的权限
- 定义列表和列表项的布局
- 定义显示联系人列表的Fragment
- 定义全局变量
- 初始化Fragment
- 为ListView设置CursorAdapter
- 设置选择联系人的监听器
- 定义Cursor的列索引常量

  对于这种形式的查询，你需要使用与在“使用特定类型的数据匹配联系人”那一节中相同的表，也可以使用相同的列索引。

- 定义onItemClick()方法
- 初始化loader
- 实现onLoadFinished()方法和onLoaderReset()方法

以下的步骤展示了为了能够根据任意的数据类型去匹配查询字符串并显示结果列表，我们需要添加的额外代码。

去除查询标准

不需要为mSelectionArgs定义查询标准常量SELECTION。这些内容在这种类型的检索不会被用到。

实现**onCreateLoader()**方法

实现onCreateLoader()方法，返回一个新的CursorLoader对象。我们不需要把搜索字符串转化成一个搜索模式，因为Contacts Provider会自动做这件事。使用Contacts.CONTENT_FILTER_URI作为基础查询URI，并使用Uri.withAppendedPath()方法将

搜索字符串添加到基础URI中。使用这个URI会自动触发对任意数据类型的搜索，就像以下例子所示：

```
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    /*
     * Appends the search string to the base URI. Always
     * encode search strings to ensure they're in proper
     * format.
     */
    Uri contentUri = Uri.withAppendedPath(
            Contacts.CONTENT_FILTER_URI,
            Uri.encode(mSearchString));
    // Starts the query
    return new CursorLoader(
            getActivity(),
            contentUri,
            PROJECTION,
            null,
            null,
            null
    );
}
```

这段代码片段，是想要在Contacts Provider中建立广泛搜索类型应用的基础部分。这种方法对那些想要实现与通讯录应用联系人列表中相似搜索功能的应用，会很有帮助。

获取联系人详情

> 编写:spencer198711 - 原文:http://developer.android.com/training/contacts-provider/retrieve-details.html

这一课展示了如何取得一个联系人的详细信息，比如email地址、电话号码等。当使用者去获取联系人信息的时候，这些信息正是他们所查找的。我们可以给他们关于一个联系人的所有信息，或者仅仅显示一个特定的数据类型，比如email地址。

这一课假设你已经获取到了一个用户所选取的联系人的ContactsContract.Contacts数据项。在获取联系人名字那一课展示了如何获取联系人列表。

获取联系人的所有详细信息

为了取得一个联系人的所有详情，查找ContactsContract.Data表中包含联系人LOOKUP_KEY列的任意行。因为Contacts Provider隐式地连接了ContactsContract.Contacts表和ContactsContract.Data表，所以这个LOOKUP_KEY列在ContactsContract.Data表中是可用的。关于LOOKUP_KEY列，在获取联系人名字那一课有详细的描述。

> **Note**：检索一个联系人的所有信息会降低设备的性能，因为这需要检索ContactsContract.Data表的所有列。在使用这种方法之前，请认真考虑对性能影响。

请求权限

为了能够读Contacts Provider，我们的应用必须拥有READ_CONTACTS权限。为了请求这个权限，需要在manifest文件的中添加如下子节点：

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

设置查询映射

根据一行数据的数据类型，它可能会使用很多列或者只使用几列。另外，数据会根据不同的数据类型而出现在不同的列中。为了确保能够获取所有数据类型的所有可能的数据列，需要在查询映射中添加所有列的名字。如果要把Cursor绑定到ListView，记得要获取Data._ID，否则的话，界面绑定就不会起作用。同时也需要获取Data.MIMETYPE列，这样才能识别我们获取到的每一行数据的数据类型。例如：

```
private static final String PROJECTION =
            {
                Data._ID,
                Data.MIMETYPE,
                Data.DATA1,
                Data.DATA2,
                Data.DATA3,
                Data.DATA4,
                Data.DATA5,
                Data.DATA6,
                Data.DATA7,
                Data.DATA8,
                Data.DATA9,
                Data.DATA10,
                Data.DATA11,
                Data.DATA12,
                Data.DATA13,
                Data.DATA14,
                Data.DATA15
            };
```

这个查询映射使用了ContactsContract.Data类中定义的列名字，去获取ContactsContract.Data表中一行的所有数据列。

我们也可以使用由ContactsContract.Data或其子类定义的列常量去设置查询映射。需要注意的是，从SYNC1到SYNC4的数据列是sync adapter同步数据所使用的，它们的值对我们没有意义。

定义查询标准

为查询选择子句定义一个常量，一个包含查询选择参数的数组，以及一个保存查询选择值的变量。使用Contacts.LOOKUP_KEY列去查找这个联系人。例如：

```
    // Defines the selection clause
    private static final String SELECTION = Data.LOOKUP_KEY + " = ?";
    // Defines the array to hold the search criteria
    private String[] mSelectionArgs = { "" };
    /*
     * Defines a variable to contain the selection value. Once you
     * have the Cursor from the Contacts table, and you've selected
     * the desired row, move the row's LOOKUP_KEY value into this
     * variable.
     */
    private String mLookupKey;
```

在查询选择表达式中使用“?”占位符，确保了搜索是由绑定生成而不是由SQL编译生成。这种方法消除了恶意SQL注入的可能性。

定义排序顺序

定义在查询结果Cursor中希望的排序顺序。按照Data.MIMETYPE去排序，可以让特定数据类型的所有行排列在一起。这种形式的查询排序参数让所有具有email的行排在一起，让所有具有电话的行排在一起......例如：

```
    /*
     * Defines a string that specifies a sort order of MIME type
     */
    private static final String SORT_ORDER = Data.MIMETYPE;
```

> **Note**：一些数据类型不使用子类型，所以不能按照子类型来排序。作为替代方法，我们不得不遍历返回的Cursor，去判定当前行的数据类型，为那些使用子类型的数据行保存数据。当读取完cursor后，我们可以根据子类型去排序每一个数据类型并显示结果。

初始化查询loader

永远在后台线程中去检索Contacts Provider(或者其他content provider)的数据。使用Loader框架中的LoaderManager类和LoaderManager.LoaderCallbacks在后台去做获取数据的工作。

当我们已经准备好去获取数据行，需要通过调用initLoader()方法去初始化loader框架。传递一个Integer类型的标识符给initLoader()方法，这个标识符会传递给LoaderManager.LoaderCallbacks方法。当在一个应用中使用多个loader时，这个标识符能够帮助我们区分它们。

以下的代码片段展示了如何初始化loader框架：

```
public class DetailsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor> {
    ...
    // Defines a constant that identifies the loader
    DETAILS_QUERY_ID = 0;
    ...
    /*
     * Invoked when the parent Activity is instantiated
     * and the Fragment's UI is ready. Put final initialization
     * steps here.
     */
    @Override
    onActivityCreated(Bundle savedInstanceState) {
        ...
        // Initializes the loader framework
        getLoaderManager().initLoader(DETAILS_QUERY_ID, null, this);
```

实现onCreateLoader()方法

实现onCreateLoader()方法。loader框架会在我们调用initLoader()方法后立即调用onCreateLoader()方法。这个方法会返回一个CursorLoader对象。由于搜索的是ContactsContract.Data表，所以需要使用常量Data.CONTENT_URI作为内容URI。例如：

```
    @Override
    public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
        // Choose the proper action
        switch (loaderId) {
            case DETAILS_QUERY_ID:
            // Assigns the selection parameter
            mSelectionArgs[0] = mLookupKey;
            // Starts the query
            CursorLoader mLoader =
                    new CursorLoader(
                            getActivity(),
                            Data.CONTENT_URI,
                            PROJECTION,
                            SELECTION,
                            mSelectionArgs,
                            SORT_ORDER
                    );
            ...
    }
```

实现onLoadFinished()方法和onLoaderReset()方法

实现onLoadFinished()方法。当Contacts Provider返回查询结果的时候，loader框架会调用onLoadFinished()方法。例如：

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
        switch (loader.getId()) {
            case DETAILS_QUERY_ID:
                    /*
                     * Process the resulting Cursor here.
                     */
                }
                break;
            ...
        }
    }
```

当loader框架检测到结果集Cursor所对应的数据已经发生变化的时候，会调用onLoaderReset()方法。这时，需要通过把Cursor设置为null来移除对已经存在Cursor对象的引用。否则，loader框架就不会销毁旧的Cursor对象，从而导致内存泄漏。例如：

```
    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        switch (loader.getId()) {
            case DETAILS_QUERY_ID:
                /*
                 * If you have current references to the Cursor,
                 * remove them here.
                 */
                }
                break;
    }
```

获取联系人的特定类型的信息

获取联系人的特定类型的信息，例如所有的email信息，跟获取联系人的所有详细信息类似。下面的内容是在获取联系人的所有详细信息列出的代码的基础上作出的修改：

查询映射

修改查询映射使得能够针对特定的数据类型去获取列。同时需要修改查询映射，来把在ContactsContract.CommonDataKinds子类中定义的列常量与数据类型对应起来。

查询选择

修改查询选择子句去搜索特定类型的MIMETYPE值。

排序顺序

由于仅仅搜索一种类型的详细数据，所以不需要将返回的Cursor按照Data.MIMETYPE进行分组。

这些修改将会在下面的小节中详细描述。

设置查询映射

使用ContactsContract.CommonDataKinds的特定类型子类所定义的列名称常量，定义我们想要获取的数据列。如果我们打算把Cursor绑定到ListView，确保要获取 `_ID` 列。例如，为了获取email数据，需要定义以下数据映射：

```
private static final String[] PROJECTION =
            {
                Email._ID,
                Email.ADDRESS,
                Email.TYPE,
                Email.LABEL
            };
```

需要注意的是，这个查询映射使用在ContactsContract.CommonDataKinds.Email类中定义的列名称，来替代ContactsContract.Data类中定义的列名称。使用email类型的列名称使得代码更具可读性。

在查询映射中，我们也可以使用ContactsContract.CommonDataKinds子类所定义的其他数据列。

定义查询标准

根据我们想要找的特定联系人的LOOKUP_KEY和联系人详细信息的Data.MIMETYPE定义一个搜索表达式，去获取数据。把MIMETYPE的值从头到尾用单引号括住，否则的话，content provider将会把这个常量当成变量名而不是字符串。因为我们使用的是常量，而不是用户提供的值，所以这里不需要使用占位符。例如：

```
/*
     * Defines the selection clause. Search for a lookup key
     * and the Email MIME type
     */
    private static final String SELECTION =
            Data.LOOKUP_KEY + " = ?" +
            " AND " +
            Data.MIMETYPE + " = " +
            "'" + Email.CONTENT_ITEM_TYPE + "'";
    // Defines the array to hold the search criteria
    private String[] mSelectionArgs = { "" };
```

定义排序规则

为查询返回的Cursor定义一个排序规则。由于是检索特定的数据类型，删除根据MIMETYPE来排序的部分。而如果查询的详细数据类型包含子类型，可以根据这个子类型去排序。例如，对于email数据，我们可以根据Email.TYPE排序：

```
private static final String SORT_ORDER = Email.TYPE + " ASC ";
```

使用Intent修改联系人信息

> 编写:spencer198711 - 原文:http://developer.android.com/training/contacts-provider/modify-data.html

这一课介绍如何使用Intent去插入一个新的联系人或者修改联系人的数据。我们不是直接访问Contacts Provider，而是通过Intent启动Contacts应用去运行适当的Activity。对于这一课中描述的数据修改行为，如果你向Intent发送扩展的数据，它会自动填充进启动的Activity页面中。

使用Intent去插入或者更新一个联系人是比较推荐的修改Contacts Provider的做法。原因如下：

- 节省了我们自行开发UI和编写代码的时间和精力。
- 避免了由于不按照Contacts Provider的规则去修改而产生的错误。
- 减少应用需要申请的权限数量。因为我们的应用把修改行为委托给已经拥有写Contacts Provider权限的Contacts应用，所以我们的应用不需要再去申请这个权限，。

使用Intent插入新的联系人

当我们的应用接收到新的数据时，我们通常会允许用户去插入一个新的联系人。例如，一个餐馆评论应用可以允许用户在评论餐馆的时候，把这个餐馆添加为一个联系人。可以使用Intent去做这个任务，使用我们拥有的尽可能多的数据去创建对应的Intent，然后发送这个Intent到Contacts应用。

使用Contacts应用去插入一个联系人将会向Contacts Provider中的ContactsContract.RawContacts表中插入一个原始联系人。必要的情况下，在创建原始联系人的时候，Contacts应用将会提示用户选择账户类型和要使用的账户。如果联系人已经存在，Contacts应用也会告知用户。用户将会有取消插入的选项，在这种情况下不会有联系人被创建。想要知道更多关于原始联系人的信息，请参阅Contacts Provider的API指导。

创建一个Intent

利用Intents.Insert.ACTION创建一个新的Intent对象，并设置其MIME类型为RawContacts.CONTENT_TYPE。例如：

```
...
// Creates a new Intent to insert a contact
Intent intent = new Intent(Intents.Insert.ACTION);
// Sets the MIME type to match the Contacts Provider
intent.setType(ContactsContract.RawContacts.CONTENT_TYPE);
```

如果我们已经获得了此联系人的详细信息，比如说电话号码或者email地址，那么我们可以把它们作为扩展数据添加到Intent中。对于键值，需要使用Intents.Insert中对应的常量。Contacts应用将会在插入界面显示这些数据，以便用户作进一步的数据编辑和数据添加。

```
/* Assumes EditText fields in your UI contain an email address
 * and a phone number.
 *
 */
private EditText mEmailAddress = (EditText) findViewById(R.id.email);
private EditText mPhoneNumber = (EditText) findViewById(R.id.phone);
...
/*
 * Inserts new data into the Intent. This data is passed to the
 * contacts app's Insert screen
 */
// Inserts an email address
intent.putExtra(Intents.Insert.EMAIL, mEmailAddress.getText())
/*
 * In this example, sets the email type to be a work email.
 * You can set other email types as necessary.
 */
      .putExtra(Intents.Insert.EMAIL_TYPE, CommonDataKinds.Email.TYPE_WORK)
// Inserts a phone number
      .putExtra(Intents.Insert.PHONE, mPhoneNumber.getText())
/*
 * In this example, sets the phone type to be a work phone.
 * You can set other phone types as necessary.
 */
      .putExtra(Intents.Insert.PHONE_TYPE, Phone.TYPE_WORK);
```

一旦我们创建好Intent，调用startActivity()将其发送到Contacts应用。

```
    /* Sends the Intent
     */
    startActivity(intent);
```

这个调用将会打开Contacts应用的界面，并允许用户进入一个新的联系人。这个联系人的账户类型和账户名字列在屏幕的上方。一旦用户输入数据并点击确定，Contacts应用的联系人列表则会显示出来。用户可以点击*Back*键返回到我们自己创建的应用。

使用Intent编辑已经存在的联系人

如果用户已经选择了一个感兴趣的联系人，使用Intent去编辑这个已存在的联系人会很有用。例如，一个用来查找拥有邮政地址但是缺少邮政编码的联系人的应用，可以给用户提供查找邮政编码的选项，然后把找到的邮政编码添加到这个联系人中。

使用Intent编辑已经存在的联系人，同插入一个联系人的步骤类似。像前面介绍的使用Intent插入新的联系人创建一个Intent，但是需要给这个Intent添加对应联系人的Contacts.CONTENT_LOOKUP_URI和MIME类型Contacts.CONTENT_ITEM_TYPE。如果想要使用已经拥有的详情信息编辑这个联系人，我们需要把这些数据放到Intent的扩展数据中。同时注意有些列是不能使用Intent编辑的，这些不可编辑的列在ContactsContract.Contacts 摘要部分“Update”标题下有列出。

最后，发送这个Intent。Contacts应用会显示一个编辑界面作为回应。当用户编辑完成并保存，Contacts应用会显示一个联系人列表。当用户点击*Back*，我们自己的应用会出现。

创建Intent

为了能够编辑一个联系人，需要调用Intent(action)去创建一个拥有ACTION_EDIT行为的Intent。调用setDataAndType()去设置这个Intent要编辑的联系人的Contacts.CONTENT_LOOKUP_URI和MIME类型Contacts.CONTENT_ITEM_TYPE。因为调用setType()会重写Intent当前的数据，所以我们必须同时设置数据和MIME类型。

为了得到联系人的Contacts.CONTENT_LOOKUP_URI，需要调用Contacts.getLookupUri(id, lookupkey)方法，该方法的参数分别是联系人的Contacts._ID和Contacts.LOOKUP_KEY。

以下的代码片段展示了如何创建这个Intent：

```
// The Cursor that contains the Contact row
    public Cursor mCursor;
    // The index of the lookup key column in the cursor
    public int mLookupKeyIndex;
    // The index of the contact's _ID value
    public int mIdIndex;
    // The lookup key from the Cursor
    public String mCurrentLookupKey;
    // The _ID value from the Cursor
    public long mCurrentId;
    // A content URI pointing to the contact
    Uri mSelectedContactUri;
    ...
    /*
     * Once the user has selected a contact to edit,
     * this gets the contact's lookup key and _ID values from the
     * cursor and creates the necessary URI.
     */
    // Gets the lookup key column index
    mLookupKeyIndex = mCursor.getColumnIndex(Contacts.LOOKUP_KEY);
    // Gets the lookup key value
    mCurrentLookupKey = mCursor.getString(mLookupKeyIndex);
    // Gets the _ID column index
    mIdIndex = mCursor.getColumnIndex(Contacts._ID);
    mCurrentId = mCursor.getLong(mIdIndex);
    mSelectedContactUri =
            Contacts.getLookupUri(mCurrentId, mCurrentLookupKey);
    ...
    // Creates a new Intent to edit a contact
    Intent editIntent = new Intent(Intent.ACTION_EDIT);
    /*
     * Sets the contact URI to edit, and the data type that the
     * Intent must match
     */
    editIntent.setDataAndType(mSelectedContactUri,Contacts.CONTENT_ITEM_TYPE);
```

添加导航标志

在Android 4.0（API版本14）和更高的版本，Contacts应用中的一个问题会导致错误的页面导航。我们的应用发送一个编辑联系人的Intent到Contacts应用，用户编辑并保存这个联系人，当用户点击*Back*键的时候会看到联系人列表页面。用户需要点击最近使用的应用，然后选择我们的应用，才能返回到我们自己的应用。

要在Android 4.0.3（API版本15）及以后的版本解决此问题，需要添加 `finishActivityOnSaveCompleted` 扩展数据参数到这个Intent，并将它的值设置为true。Android 4.0之前的版本也能够接受这个参数，但是不起作用。为了设置扩展数据，请按照以下方式去做：

```
    // Sets the special extended data for navigation
    editIntent.putExtra("finishActivityOnSaveCompleted", true);
```

添加其他的扩展数据

对Intent添加额外的扩展数据，需要调用putExtra()。可以为常见的联系人数据字段添加扩展数据，这些常见字段的key值可以从Intents.Insert API参考文档中查到。记住ContactsContract.Contacts表中有些列是不能编辑的，这些列在ContactsContract.Contacts的摘要部分“Update”标题下有列出。

发送Intent

最后，发送我们已经构建好的Intent。例如：

```
    // Sends the Intent
    startActivity(editIntent);
```

使用Intent让用户去选择是插入还是编辑联系人

我们可以通过发送带有 `ACTION_INSERT_OR_EDIT` 行为的Intent，让用户去选择是插入联系人还是编辑已有的联系人。例如，一个email客户端应用会允许用户添加一个收件地址到新的联系人，或者仅仅作为额外的邮件地址添加到已有的联系人。需要为这个Intent设置MIME类型Contacts.CONTENT_ITEM_TYPE，但是不需要设置数据URI。

当我们发送这个Intent后，Contacts应用会展示一个联系人列表。用户可以选择是插入一个新的联系人还是挑选一个存在的联系人去编辑。任何添加到Intent中的扩展数据字段都会填充在界面上。我们可以使用任何在Intents.Insert中指定的的key值。以下的代码片段展示了如何构建和发送这个Intent：

```
// Creates a new Intent to insert or edit a contact
    Intent intentInsertEdit = new Intent(Intent.ACTION_INSERT_OR_EDIT);
    // Sets the MIME type
    intentInsertEdit.setType(Contacts.CONTENT_ITEM_TYPE);
    // Add code here to insert extended data, if desired
    ...
    // Sends the Intent with an request ID
    startActivity(intentInsertEdit);
```

显示联系人头像

> 编写:spencer198711 - 原文:http://developer.android.com/training/contacts-provider/display-contact-badge.html

这一课展示了如何在我们的应用界面上添加一个QuickContactBadge，以及如何为它绑定数据。QuickContactBadge是一个在初始情况下显示联系人缩略图头像的widget。尽管我们可以使用任何Bitmap作为缩略图头像，但是我们通常会使用从联系人照片缩略图中解码出来的Bitmap。

这个小的图片是一个控件，当用户点击它时，QuickContactBadge会展开一个包含以下内容的对话框：

- 一个大的联系人头像

  与这个联系人关联的大的头像，如果此人没有设置头像，则显示预留的图案。

- 应用程序图标

  根据联系人详情数据，显示每一个能够被手机中的应用所处理的数据的图标。例如，如果联系人的数据包含一个或多个email地址，就会显示email应用的图标。当用户点击这个图标的时候，这个联系人所有的email地址都会显示出来。当用户点击其中一个email地址时，email应用将会显示一个界面，让用户为选中的地址撰写邮件。

QuickContactBadge视图提供了对联系人数据的即时访问，是一种与联系人沟通的快捷方式。用户不用查询一个联系人，查找并复制信息，然后把信息粘贴到合适的应用中。他们可以点击QuickContactBadge，选择他们想要的沟通方式，然后直接把信息发送给合适的应用中。

添加一个QuickContactBadge视图

为了添加一个QuickContactBadge视图，需要在布局文件中插入一个QuickContactBadge。例如：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="match_parent"
                android:layout_height="match_parent">
...
    <QuickContactBadge
               android:id=@+id/quickbadge
               android:layout_height="wrap_content"
               android:layout_width="wrap_content"
               android:scaleType="centerCrop"/>
    ...
</RelativeLayout>
```

获取Contacts Provider的数据

为了能在QuickContactBadge中显示联系人，我们需要这个联系人的内容URI和显示头像的Bitmap。我们可以从在Contacts Provider中获取到的数据列中生成这两个数据。需要指定这些列作为查询映射去把数据加载到Cursor中。

对于Android 3.0（API版本为11）以及以后的版本，需要在查询映射中添加以下列：

- Contacts._ID
- Contacts.LOOKUP_KEY
- Contacts.PHOTO_THUMBNAIL_URI

对于Android 2.3.3（API版本为10）以及之前的版本，则使用以下列：

- Contacts._ID
- Contacts.LOOKUP_KEY

这一课的剩余部分假设你已经获取到了包含这些以及其他你可能选择的数据列的Cursor对象。想要学习如何获取这些列对象的Cursor，请参阅课程获取联系人列表。

设置联系人URI和缩略图

一旦我们已经拥有了所需的数据列，那么我们就可以为QuickContactBadge视图绑定数据了。

设置联系人URI

为了设置联系人URI，需要调用getLookupUri(id, lookupKey)去获取CONTENT_LOOKUP_URI，然后调用assignContactUri())去为QuickContactBadge设置对应的联系人。例如：

```
// The Cursor that contains contact rows
Cursor mCursor;
// The index of the _ID column in the Cursor
int mIdColumn;
// The index of the LOOKUP_KEY column in the Cursor
int mLookupKeyColumn;
// A content URI for the desired contact
Uri mContactUri;
// A handle to the QuickContactBadge view
QuickContactBadge mBadge;
...
mBadge = (QuickContactBadge) findViewById(R.id.quickbadge);
/*
 * Insert code here to move to the desired cursor row
 */
// Gets the _ID column index
mIdColumn = mCursor.getColumnIndex(Contacts._ID);
// Gets the LOOKUP_KEY index
mLookupKeyColumn = mCursor.getColumnIndex(Contacts.LOOKUP_KEY);
// Gets a content URI for the contact
mContactUri =
        Contacts.getLookupUri(
            mCursor.getLong(mIdColumn),
            mCursor.getString(mLookupKeyColumn)
        );
mBadge.assignContactUri(mContactUri);
```

当用户点击QuickContactBadge图标的时候，这个联系人的详细信息将会自动展现在对话框中。

设置联系人照片的缩略图

为QuickContactBadge设置联系人URI并不会自动加载联系人的缩略图照片。为了加载联系人照片，需要从联系人的Cursor对象的一行数据中获取照片的URI，使用这个URI去打开包含压缩的缩略图文件，并把这个文件读到Bitmap对象中。

> **Note**：PHOTO_THUMBNAIL_URI这一列在Android 3.0之前的版本是不存在的。对于这些版本，我们必须从Contacts.Photo表中获取照片的URI。

首先，为包含Contacts._ID和Contacts.LOOKUP_KEY的Cursor数据列设置对应的变量，这在之前已经有描述：

```
// The column in which to find the thumbnail ID
int mThumbnailColumn;
/*
 * The thumbnail URI, expressed as a String.
 * Contacts Provider stores URIs as String values.
 */
String mThumbnailUri;
...
/*
 * Gets the photo thumbnail column index if
 * platform version >= Honeycomb
 */
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    mThumbnailColumn =
            mCursor.getColumnIndex(Contacts.PHOTO_THUMBNAIL_URI);
// Otherwise, sets the thumbnail column to the _ID column
} else {
    mThumbnailColumn = mIdColumn;
}
/*
 * Assuming the current Cursor position is the contact you want,
 * gets the thumbnail ID
 */
mThumbnailUri = mCursor.getString(mThumbnailColumn);
...
```

定义一个方法，使用与这个联系人的照片有关的数据和目标视图的尺寸作为参数，返回一个尺寸合适的缩略图Bitmap对象。下面先构建一个指向这个缩略图的URI：

```
 /**
 * Load a contact photo thumbnail and return it as a Bitmap,
 * resizing the image to the provided image dimensions as needed.
 * @param photoData photo ID Prior to Honeycomb, the contact's _ID value.
 * For Honeycomb and later, the value of PHOTO_THUMBNAIL_URI.
 * @return A thumbnail Bitmap, sized to the provided width and height.
 * Returns null if the thumbnail is not found.
 */
private Bitmap loadContactPhotoThumbnail(String photoData) {
    // Creates an asset file descriptor for the thumbnail file.
    AssetFileDescriptor afd = null;
    // try-catch block for file not found
    try {
        // Creates a holder for the URI.
        Uri thumbUri;
        // If Android 3.0 or later
        if (Build.VERSION.SDK_INT
                >=
            Build.VERSION_CODES.HONEYCOMB) {
            // Sets the URI from the incoming PHOTO_THUMBNAIL_URI
            thumbUri = Uri.parse(photoData);
```

```
        } else {
        // Prior to Android 3.0, constructs a photo Uri using _ID
            /*
             * Creates a contact URI from the Contacts content URI
             * incoming photoData (_ID)
             */
            final Uri contactUri = Uri.withAppendedPath(
                    Contacts.CONTENT_URI, photoData);
            /*
             * Creates a photo URI by appending the content URI of
             * Contacts.Photo.
             */
            thumbUri =
                    Uri.withAppendedPath(
                            contactUri, Photo.CONTENT_DIRECTORY);
        }

    /*
     * Retrieves an AssetFileDescriptor object for the thumbnail
     * URI
     * using ContentResolver.openAssetFileDescriptor
     */
    afd = getActivity().getContentResolver().
            openAssetFileDescriptor(thumbUri, "r");
    /*
     * Gets a file descriptor from the asset file descriptor.
     * This object can be used across processes.
     */
    FileDescriptor fileDescriptor = afd.getFileDescriptor();
    // Decode the photo file and return the result as a Bitmap
    // If the file descriptor is valid
    if (fileDescriptor != null) {
        // Decodes the bitmap
        return BitmapFactory.decodeFileDescriptor(
                fileDescriptor, null, null);
        }
    // If the file isn't found
    } catch (FileNotFoundException e) {
        /*
         * Handle file not found errors
         */
    }
    // In all cases, close the asset file descriptor
    } finally {
        if (afd != null) {
            try {
                afd.close();
            } catch (IOException e) {}
        }
    }
    return null;
}
```

在代码中调用loadContactPhotoThumbnail()去获取缩略图Bitmap对象，使用获取的Bitmap对象去设置QuickContactBadge头像缩略图。

```
...
/*
 * Decodes the thumbnail file to a Bitmap.
 */
Bitmap mThumbnail =
        loadContactPhotoThumbnail(mThumbnailUri);
/*
 * Sets the image in the QuickContactBadge
 * QuickContactBadge inherits from ImageView, so
 */
mBadge.setImageBitmap(mThumbnail);
```

把QuickContactBadge添加到ListView

QuickContactBadge对于一个展示联系人列表的ListView来说是一个非常有用的添加功能。使用QuickContactBadge去为每一个联系人显示一个缩略图，当用户点击这个缩略图时，QuickContactBadge对话框将会显示。

为ListView添加QuickContactBadge

首先，在列表项布局文件中添加QuickContactBadge视图元素。例如，如果我们想为获取到的每一个联系人显示QuickContactBadge和名字，把以下的XML内容放到对应的布局文件中：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="match_parent"
                android:layout_height="wrap_content">
    <QuickContactBadge
        android:id="@+id/quickcontact"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:scaleType="centerCrop"/>
    <TextView android:id="@+id/displayname"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:layout_toRightOf="@+id/quickcontact"
              android:gravity="center_vertical"
              android:layout_alignParentRight="true"
              android:layout_alignParentTop="true"/>
</RelativeLayout>
```

在以下的章节中，这个文件被称为 `contact_item_layout.xml` 。

设置自定义的CursorAdapter

定义一个继承自CursorAdapter的adapter来将CursorAdapter绑定到一个包含QuickContactBadge的ListView中。这种方式允许我们在绑定数据到QuickContactBadge之前对Cursor中的数据进行处理。同时也能将多个Cursor中的列绑定到QuickContactBadge。而使用普通的CursorAdapter是不能完成这些操作的。

我们定义的CursorAdapter的子类必须重写以下方法：

- CursorAdapter.newView()

  填充一个View对象去持有列表项布局。在重写这个方法的过程中，需要保存这个布局的子View的handles，包括QuickContactBadge的handles。通过采用这种方法，避免了每次在填充新的布局时都去获取子View的handles。

  我们必须重写这个方法以便能够获取每个子View对象的handles。这种方法允许我们控制这些子View对象在CursorAdapter.bindView()方法中的绑定。

- CursorAdapter.bindView()

  将数据从当前Cursor行绑定到列表项布局的子View对象中。必须重写这个方法以便能够将联系人的URI和缩略图信息绑定到QuickContactBadge。这个方法的默认实现仅仅允许在数据列和View之间的一对一映射。

以下的代码片段是一个包含了自定义CursorAdapter子类的例子。

定义自定义的列表Adapter

定义CursorAdapter的子类包括编写这个类的构造方法，以及重写newView()和bindView():

```
private class ContactsAdapter extends CursorAdapter {
    private LayoutInflater mInflater;
    ...
    public ContactsAdapter(Context context) {
        super(context, null, 0);

        /*
         * Gets an inflater that can instantiate
         * the ListView layout from the file.
         */
        mInflater = LayoutInflater.from(context);
        ...
    }
    ...
    /**
     * Defines a class that hold resource IDs of each item layout
     * row to prevent having to look them up each time data is
     * bound to a row.
```

```
     */
    private class ViewHolder {
        TextView displayname;
        QuickContactBadge quickcontact;
    }
    ..
    @Override
    public View newView(
            Context context,
            Cursor cursor,
            ViewGroup viewGroup) {
        /* Inflates the item layout. Stores resource IDs in a
         * in a ViewHolder class to prevent having to look
         * them up each time bindView() is called.
         */
        final View itemView =
                mInflater.inflate(
                        R.layout.contact_list_layout,
                        viewGroup,
                        false
                );
        final ViewHolder holder = new ViewHolder();
        holder.displayname =
                (TextView) view.findViewById(R.id.displayname);
        holder.quickcontact =
                (QuickContactBadge)
                        view.findViewById(R.id.quickcontact);
        view.setTag(holder);
        return view;
    }
    ...
    @Override
    public void bindView(
            View view,
            Context context,
            Cursor cursor) {
        final ViewHolder holder = (ViewHolder) view.getTag();
        final String photoData =
                cursor.getString(mPhotoDataIndex);
        final String displayName =
                cursor.getString(mDisplayNameIndex);
        ...
        // Sets the display name in the layout
        holder.displayname = cursor.getString(mDisplayNameIndex);
        ...
        /*
         * Generates a contact URI for the QuickContactBadge.
         */
        final Uri contactUri = Contacts.getLookupUri(
                cursor.getLong(mIdIndex),
                cursor.getString(mLookupKeyIndex));
        holder.quickcontact.assignContactUri(contactUri);
        String photoData = cursor.getString(mPhotoDataIndex);
```

```
        /*
         * Decodes the thumbnail file to a Bitmap.
         * The method loadContactPhotoThumbnail() is defined
         * in the section "Set the Contact URI and Thumbnail"
         */
        Bitmap thumbnailBitmap =
                loadContactPhotoThumbnail(photoData);
        /*
         * Sets the image in the QuickContactBadge
         * QuickContactBadge inherits from ImageView
         */
        holder.quickcontact.setImageBitmap(thumbnailBitmap);
}
```

设置变量

在代码中，设置相关变量，添加一个包括必须数据列的Cursor。

> **Note**：以下的代码片段使用了方法 `loadContactPhotoThumbnail()`，这个方法是在设置联系人URI和缩略图那一节中定义的。

例如：

```
public class ContactsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor> {
...
// Defines a ListView
private ListView mListView;
// Defines a ContactsAdapter
private ContactsAdapter mAdapter;
...
// Defines a Cursor to contain the retrieved data
private Cursor mCursor;
/*
 * Defines a projection based on platform version. This ensures
 * that you retrieve the correct columns.
 */
private static final String[] PROJECTION =
        {
            Contacts._ID,
            Contacts.LOOKUP_KEY,
            (Build.VERSION.SDK_INT >=
             Build.VERSION_CODES.HONEYCOMB) ?
                    Contacts.DISPLAY_NAME_PRIMARY :
                    Contacts.DISPLAY_NAME
            (Build.VERSION.SDK_INT >=
             Build.VERSION_CODES.HONEYCOMB) ?
                    Contacts.PHOTO_THUMBNAIL_ID :
                    /*
                     * Although it's not necessary to include the
```

```
                     * column twice, this keeps the number of
                     * columns the same regardless of version
                     */
                    Contacts_ID
            ...
        };
/*
 * As a shortcut, defines constants for the
 * column indexes in the Cursor. The index is
 * 0-based and always matches the column order
 * in the projection.
 */
// Column index of the _ID column
private int mIdIndex = 0;
// Column index of the LOOKUP_KEY column
private int mLookupKeyIndex = 1;
// Column index of the display name column
private int mDisplayNameIndex = 3;
/*
 * Column index of the photo data column.
 * It's PHOTO_THUMBNAIL_URI for Honeycomb and later,
 * and _ID for previous versions.
 */
private int mPhotoDataIndex =
        Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
        3 :
        0;
...
```

设置ListView

在Fragment.onCreate())方法中，实例化自定义的adapter对象，获得一个ListView的handle。

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    /*
     * Instantiates the subclass of
     * CursorAdapter
     */
    ContactsAdapter mContactsAdapter =
            new ContactsAdapter(getActivity());
    /*
     * Gets a handle to the ListView in the file
     * contact_list_layout.xml
     */
    mListView = (ListView) findViewById(R.layout.contact_list_layout);
    ...
}
...
```

在onActivityCreated())方法中，将ContactsAdapter绑定到ListView。

```
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    ...
    // Sets up the adapter for the ListView
    mListView.setAdapter(mAdapter);
    ...
}
...
```

当获取到一个包含联系人数据的Cursor时（通常在onLoadFinished()的时候），调用swapCursor()把Cursor中的数据绑定到ListView。这将会为联系人列表中的每一项都显示一个QuickContactBadge。

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    // When the loader has completed, swap the cursor into the adapter.
    mContactsAdapter.swapCursor(cursor);
}
```

当我们使用CursorAdapter或其子类中将Cursor中的数据绑定到ListView，并且使用了CursorLoader去加载Cursor数据时，记得要在onLoaderReset()方法的实现中清理对Cursor对象的引用。例如：

```
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    // Removes remaining reference to the previous Cursor
    mContactsAdapter.swapCursor(null);
}
```

Android位置信息

> 编写:penkzhou - 原文:http://developer.android.com/training/location/index.html

位置感知是移动应用一个独特的功能。用户去到哪里都会带着他们的移动设备，而将位置感知功能添加到我们的应用里，可以让用户有更加真实的情境体验。位置服务API集成在Google Play服务里面，这便于我们将自动位置跟踪、地理围栏和用户活动识别等位置感知功能添加到我们的应用当中。

我们喜欢用Google Play services location APIs胜过Android framework location APIs (android.location) 来给我们的应用添加位置感知功能。如果你现在正在使用Android framework location APIs，我们强烈建议你尽可能切换到Google Play services location APIs。

这个课程介绍如何使用Google Play services location APIs来获取当前位置、周期性地更新位置以及查找地址。创建并监视地理围栏以及探测用户的活动。这个课程包括示例应用和代码片段，你可以利用这些资源作为添加位置感知到你的应用的基础。

> **Note**：因为这个课程基于Google Play services client library，所以在使用这些示例应用和代码段之前确保你安装了最新版本的Google Play services client library。要想学习如何安装最新版的client library，请参考安装Google Play services向导。

Lessons

- 获取最后可知位置

  学习如何获取Android设备的最后可知位置。通常Android设备的最后可知位置相当于用户的当前位置。

- 接收位置更新

  学习如何请求和接收周期性的位置更新。

- 显示位置地址

  学习如何将一个位置的经纬度转化成一个地址（反向地理编码）。

- 创建和监视地理围栏

  学习如何将一个或多个地理区域定义成一个兴趣位置集合，称为地理围栏。学习如何探测用户靠近或者进入地理围栏事件。

获取最后可知位置

> 编写:penkzhou - 原文:http://developer.android.com/training/location/retrieve-current.html

使用Google Play services location APIs，我们的应用可以请求获得用户设备的最后可知位置。大多数情况下，我们会对用户的当前位置比较感兴趣。而通常用户的当前位置相当于设备的最后可知位置。

特别地，使用fused location provider来获取设备的最后可知位置。fused location provider是Google Play services location APIs中的一个。它处理基本定位技术并提供一个简单的API，使得我们可以指定高水平的需求，如高精度或者低功耗。同时它优化了设备的耗电情况。

这节课介绍如何通过使用fused location provider的getLastLocation())方法为设备的位置构造一个单一请求。

安装Google Play Services

为了访问fused location provider，我们的应用开发工程必须包括Google Play services。通过SDK Manager下载和安装Google Play services组件，添加相关的库到我们的工程。更详细的介绍，请看Setting Up Google Play Services。

确定应用的权限

使用位置服务的应用必须请求用户位置权限。Android拥有两种位置权限：ACCESS_COARSE_LOCATION 和 ACCESS_FINE_LOCATION。我们选择的权限决定API返回的位置信息的精度。如果我们选择了ACCESS_COARSE_LOCATION，API返回的位置信息的精确度大体相当于一个城市街区。

这节课只要求粗略的定位。在我们应用的manifest文件中，用 `uses-permission` 节点请求这个权限，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.basiclocationsample" >

  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
</manifest>
```

连接Google Play Services

为了连接到API，我们需要创建一个Google Play services API客户端实例。关于使用这个客户端的更详细的介绍，请看Accessing Google APIs。

在我们的activity的onCreate()方法中，用GoogleApiClient.Builder创建一个Google API Client实例。使用这个builder添加LocationServices API。

实例应用定义了一个 `buildGoogleApiClient()` 方法，这个方法在activity的onCreate()方法中被调用。 `buildGoogleApiClient()` 方法包括下面的代码。

```
protected synchronized void buildGoogleApiClient() {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
}
```

获取最后可知位置

一旦我们将Google Play services和location services API连接完成后，我们就可以获取用户设备的最后可知位置。当我们的应用连接到这些服务之后，我们可以用fused location provider的getLastLocation()方法来获取设备的位置。调用这个方法返回的定位精确度是由我们在应用的manifest文件里添加的权限决定的，如本文的确定应用的权限部分描述的内容一样。

为了请求最后可知位置，调用getLastLocation()方法，并将我们创建的GoogleApiClient对象的实例传给该方法。在Google API Client提供的onConnected()回调函数里调用getLastLocation()方法，这个回调函数在client准备好的时候被调用。下面的示例代码说明了请求和一个对响应简单的处理：

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        mLastLocation = LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);
        if (mLastLocation != null) {
            mLatitudeText.setText(String.valueOf(mLastLocation.getLatitude()));
            mLongitudeText.setText(String.valueOf(mLastLocation.getLongitude()));
        }
    }
}
```

getLastLocation())方法返回一个Location对象。通过Location对象，我们可以取得地理位置的经度和纬度坐标。在少数情况下，当位置不可用时，这个Location对象会返回null。

下一课，获取位置更新，教你如何周期性地获取位置信息更新。

获取位置更新

> 编写:penkzhou - 原文:http://developer.android.com/training/location/receive-location-updates.html

如果我们的应用可以周期性地跟踪位置，那么应用可以给用户提供更多相关信息。例如，如果我们的应用在用户行走或者驾车时帮助找到他们的路，或者如果我们的应用跟踪用户的位置，那么它需要定期获取设备的位置。除了地理位置之外（经度和纬度），我们可能还想为用户提供更多的信息，例如方位（行驶的水平方向）、海拔或者设备的速度。这些信息可以在 Location 对象中获得，我们的应用可以从 fused location provider 中得到这个对象。

当我们用 getLastLocation()) 获取设备的位置时，如上一节课获取最后可知位置介绍的一样，一个更加直接的方法是从 fused location provider 中请求周期性的更新。作为回应，API根据现有的位置供应源，如Wifi和GPS（Global Positioning System），用最佳位置周期地更新我们的应用。这些providers、我们请求的权限和我们在位置请求中设置的选项决定了位置的精确度。

这节课介绍如何用fused location provider的requestLocationUpdates()方法来请求定期更新设备的位置。

连接Location Services

应用的 Location Services 由 Google Play services 和 fused location provider 提供。为了用这些服务，用 Google API Client 连接到我们的应用，然后请求位置更新。用 GoogleApiClient 进行连接的详细步骤请见获取最后可知位置，包括了请求当前位置。

设备的最后可知位置提供有关起点的基准信息，在开始定期更新位置信息前，保证应用拥有一个可知的位置。获取最后可知位置介绍了如何通过调用 getLastLocation()) 获取最后可知位置。接下来的内容假设我们的应用已经取得最后可知位置，并已将最后可知位置作为一个 Location 对象保存在全局变量 `mCurrentLocation` 中。

使用位置服务的应用必须请求位置权限。在这节课中我们需要很好的定位检测，使得我们的应用可以从可用的位置供应源得到尽可能精确的位置数据。在我们应用的manifest文件中，用 `uses-permission` 节点请求位置权限，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationupdates" >

  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
</manifest>
```

设置位置请求

创建一个 LocationRequest 以保存请求 fused location provider 的参数。这些参数决定了请求精确度的水平。对于位置请求中所有可用的选项，请见 LocationRequest 类的参考文档。这节课设置更新间隔、最快更新间隔和优先级。如下所述：

更新间隔

setInterval()) - 这个方法设置应用接收位置更新的速率（每毫秒）。注意如果另一个应用正在接收一个更快的或者更慢的更新速率，又或者根本没有更新（例如，设备还没有连接），那么我们应用的位置更新速率可能会比 `setInterval()` 设置的速率更快。

最快更新间隔

setFastestInterval()) - 这个方法设置应用可以处理位置更新的最快速率（每毫秒）。因为其它应用会影响到已发送出去的位置更新的速率，所以我们需要设置这个最快速率。Google Play services location APIs 发送任何应用用 setInterval()) 请求的最快的更新速率。如果这个速率比我们的应用可以处理的速率还要快，那么我们可能会遇到UI闪烁或者数据溢出等问题。为了避免这个问题，调用 setFastestInterval()) 限制更新速率的上限。

优先级

setPriority()) - 这个方法设置请求的优先级，为 Google Play services 位置服务提供了关于使用哪个位置源的强烈的暗示。支持下面几个值：

- PRIORITY_BALANCED_POWER_ACCURACY - 这个设置请求一个城市街区范围的位置精确度（精确度约为100米）。这被认为是一个粗略的精确度，也可能是耗电较小的设置。对于这个设置，位置服务可能使用 WiFi 和基站进行定位。注意，无论如何，位置供应源的选择依赖于很多其它的因素。
- PRIORITY_HIGH_ACCURACY - 这个设置请求最高精度的位置信息。对于这个设置，位置服务更可能使用 GPS(Global Positioning System) 来定位。
- PRIORITY_LOW_POWER - 这个设置请求一个城市范围的精确度（精确度约为10公里）。这被认为是一个粗略的精确度，也可能是耗电较小的设置。
- PRIORITY_NO_POWER - 如果需要对功率消耗的影响微乎其微，但又想在可用的时候接收位置更新，那么使用这个设置。对于这个设置，我们的应用不会触发任何位置更新，但是会接收由其它应用触发的位置。

下面的示例介绍创建位置请求和设置相关的参数：

```
protected void createLocationRequest() {
    LocationRequest mLocationRequest = new LocationRequest();
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
}
```

PRIORITY_HIGH_ACCURACY 的优先级联合在我们应用的 manifest 文件中定义的 ACCESS_FINE_LOCATION 权限和一个5000毫秒（5秒）的更新间隔。该优先级使 fused location provider 返回精确到几英尺之内的位置更新。这个方法适用于需要实时显示位置的地图应用。

> 性能提示：如果我们的应用在接收一个位置更新后接入网络或者执行持续时间长的工作，那么将最快更新间隔调整到一个更慢的值。这个调整防止我们的应用接收不可用的更新。一旦持续时间长的工作完成，将最快更新间隔改回一个快的值。

请求位置更新

我们已经设置了包含应用位置更新要求的位置请求，我们可以调用 requestLocationUpdates() 来启动周期性的更新。在 Google API Client 提供的 onConnected()) 回调函数（当 client 准备好之后会调用这个回调函数）中启动周期性更新。

根据请求的形式，fused location provider 要么调用 LocationListener.onLocationChanged() 回调函数并传递一个 Location 对象，要么发出一个将位置信息包含在扩展数据的 PendingIntent。更新的精确度和频率受已请求的位置权限和在位置请求对象中设置的选项等因素影响。

这节课介绍如何使用 LocationListener 回调函数获取位置更新。调用 requestLocationUpdates() ,并传入 GoogleApiClient 的实例、LocationRequest 对象和一个 LocationListener。定义一个 `startLocationUpdates()` 方法，该方法在 onConnected()) 回调函数被调用，如下面的示例代码所示：

```
@Override
public void onConnected(Bundle connectionHint) {
    ...
    if (mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}

protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates(
            mGoogleApiClient, mLocationRequest, this);
}
```

注意到上述的代码片段提到一个布尔标志位，`mRequestingLocationUpdates`，该标志位用于判断用户将位置更新打开还是关闭。关于这个标志位更详细的介绍，请见下面的保存 Activity 的状态的内容。

定义位置更新回调函数

fused location provider 调用 LocationListener.onLocationChanged()) 回调函数。这个回调函数传入的参数是一个含有位置经纬度的 Location 对象。下面的代码介绍了如何实现 LocationListener 接口和定义方法，然后获取位置更新的时间戳并在应用用户界面上显示经度、纬度和时间戳：

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener, LocationListener {
    ...
    @Override
    public void onLocationChanged(Location location) {
        mCurrentLocation = location;
        mLastUpdateTime = DateFormat.getTimeInstance().format(new Date());
        updateUI();
    }

    private void updateUI() {
        mLatitudeTextView.setText(String.valueOf(mCurrentLocation.getLatitude()));
        mLongitudeTextView.setText(String.valueOf(mCurrentLocation.getLongitude()));
        mLastUpdateTimeTextView.setText(mLastUpdateTime);
    }
}
```

停止位置更新

我们需要考虑当 activity 不在焦点上时我们是否需要停止位置更新，例如，当用户切换到另一个应用或者同一个应用的不同 activity 的情况。假如应用即使在后台运行时也不需要收集用户数据，将会有利于降低功耗。这节课会介绍如何在 activity 的 onPause()) 方法里停止位置更新。

为了停止位置更新，调用 removeLocationUpdates()，并传入 GoogleApiClient 对象的实例和一个 LocationListener，如下面的示例代码所示：

```
@Override
protected void onPause() {
    super.onPause();
    stopLocationUpdates();
}

protected void stopLocationUpdates() {
    LocationServices.FusedLocationApi.removeLocationUpdates(
            mGoogleApiClient, this);
}
```

使用一个布尔值， `mRequestingLocationUpdates` ，来判断当前位置更新是否打开。在 activity 的 onResume()) 方法里，检查当前的位置更新是否起作用。如果位置更新不起作用，那么激活它：

```
@Override
public void onResume() {
    super.onResume();
    if (mGoogleApiClient.isConnected() && !mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}
```

保存 Activity 的状态

一个设备配置的变动，如旋转屏幕或者改变语言，可以导致当前的 activity 崩溃。我们的应用必须保存任何在重新创建 activity 时需要用到的信息。一种方法是通过一个保存在 Bundle 对象的实例状态来解决这个问题。

下面的示例代码介绍了如何用 activity 的 onSaveInstanceState()) 回调函数来保存实例状态：

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putBoolean(REQUESTING_LOCATION_UPDATES_KEY,
            mRequestingLocationUpdates);
    savedInstanceState.putParcelable(LOCATION_KEY, mCurrentLocation);
    savedInstanceState.putString(LAST_UPDATED_TIME_STRING_KEY, mLastUpdateTime);
    super.onSaveInstanceState(savedInstanceState);
}
```

定义一个 `updateValuesFromBundle()` 方法来恢复保存在 activity 的上一个实例的值（如果这些值可用的话）。在 onCreate()) 中调用这个方法。如下所示：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    updateValuesFromBundle(savedInstanceState);
}

private void updateValuesFromBundle(Bundle savedInstanceState) {
    if (savedInstanceState != null) {
        // Update the value of mRequestingLocationUpdates from the Bundle, and
        // make sure that the Start Updates and Stop Updates buttons are
        // correctly enabled or disabled.
        if (savedInstanceState.keySet().contains(REQUESTING_LOCATION_UPDATES_KEY)) {
            mRequestingLocationUpdates = savedInstanceState.getBoolean(
                    REQUESTING_LOCATION_UPDATES_KEY);
            setButtonsEnabledState();
        }

        // Update the value of mCurrentLocation from the Bundle and update the
        // UI to show the correct latitude and longitude.
        if (savedInstanceState.keySet().contains(LOCATION_KEY)) {
            // Since LOCATION_KEY was found in the Bundle, we can be sure that
            // mCurrentLocationis not null.
            mCurrentLocation = savedInstanceState.getParcelable(LOCATION_KEY);
        }

        // Update the value of mLastUpdateTime from the Bundle and update the UI.
        if (savedInstanceState.keySet().contains(LAST_UPDATED_TIME_STRING_KEY)) {
            mLastUpdateTime = savedInstanceState.getString(
                    LAST_UPDATED_TIME_STRING_KEY);
        }
        updateUI();
    }
}
```

更多关于保存实例状态的内容，请看 Android Activity 类的参考文档。

> **Note**：为了可以更加持久地存储，我们可以将用户的偏好设定保存在应用的 SharedPreferences 中。在 activity 的 onPause()) 方法中设置偏好设定，在 onResume()) 中获取这些设定。更多关于偏好设定的内容，请见保存到 Rreference。

下一节课，显示位置地址，介绍如何显示指定位置的街道地址。

显示位置地址

> 编写:penkzhou - 原文:http://developer.android.com/training/location/display-address.html

获取最后可知位置和获取位置更新课程描述了如何以一个Location对象的形式获取用户的位置信息，这个位置信息包括了经纬度。尽管经纬度对计算地理距离和在地图上显示位置很有用，但是更多情况下位置的地址更有用。例如，如果我们想让用户知道他们在哪里，那么一个街道地址比地理坐标（经度/纬度）更加有意义。

使用 Android 框架位置 APIs 的 Geocoder 类，我们可以将地址转换成相应的地理坐标。这个过程叫做地理编码。或者，我们可以将地理位置转换成相应的地址。这种地址查找功能叫做反向地理编码。

这节课介绍了如何用 getFromLocation() 方法将地理位置转换成地址。这个方法返回与制定经纬度相对应的估计的街道地址。

获取地理位置

设备的最后可知位置对于地址查找功能是很有用的基础。获取最后可知位置介绍了如何通过调用 fused location provider 提供的 getLastLocation()) 方法找到设备的最后可知位置。

为了访问 fused location provider，我们需要创建一个 Google Play services API client 的实例。关于如何连接 client，请见连接 Google Play Services 。

为了让 fused location provider 得到一个准确的街道地址，在应用的 manifest 文件添加位置权限 `ACCESS_FINE_LOCATION` ，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationupdates" >

  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
</manifest>
```

定义一个 Intent 服务来取得地址

Geocoder 类的 getFromLocation() 方法接收一个经度和纬度，返回一个地址列表。这个方法是同步的，可能会花很长时间来完成它的工作，所以我们不应该在应用的主线程和 UI 线程里调用这个方法。

IntentService 类提供了一种结构使一个任务在后台线程运行。使用这个类，我们可以在不影响 UI 响应速度的情况下处理一个长时间运行的操作。注意到，AsyncTask 类也可以执行后台操作，但是它被设计用于短时间运行的操作。在 activity 重新创建时（例如当设备旋转时），AsyncTask 不应该保存 UI 的引用。相反，当 activity 重建时，不需要取消 IntentService。

定义一个继承 IntentService 的类 `FetchAddressIntentService`。这个类是地址查找服务。这个 Intent 服务在一个工作线程上异步地处理一个 intent，并在它离开这个工作时自动停止。Intent 外加的数据提供了服务需要的数据，包括一个用于转换成地址的 Location 对象和一个用于处理地址查找结果的 ResultReceiver 对象。这个服务用一个 Geocoder 来获取位置的地址，并且将结果发送给 ResultReceiver。

在应用的 manifest 文件中定义 Intent 服务

在 manifest 文件中添加一个节点以定义 intent 服务：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationaddress" >
    <application
        ...
        <service
            android:name=".FetchAddressIntentService"
            android:exported="false"/>
    </application>
    ...
</manifest>
```

> **Note**：manifest 文件里的 `<service>` 节点不需要包含一个 intent filter，这是因为我们的主 activity 通过指定 intent 用到的类的名字来创建一个隐式的 intent。

创建一个 Geocoder

将一个地理位置传换成地址的过程叫做反向地理编码。通过实现 `FetchAddressIntentService` 类的 onHandleIntent()) 来执行 intent 服务的主要工作，即反向地理编码请求。创建一个 Geocoder 对象来处理反向地理编码。

一个区域设置代表一个特定的地理上的或者语言上的区域。Locale 对象用于调整信息的呈现方式，例如数字或者日期，来适应区域设置表示的区域的约定。传一个 Locale 对象到 Geocoder 对象，确保地址结果为用户的地理区域作出了本地化。

```
@Override
protected void onHandleIntent(Intent intent) {
    Geocoder geocoder = new Geocoder(this, Locale.getDefault());
    ...
}
```

获取街道地址数据

下一步是从 geocoder 获取街道地址，处理可能出现的错误，和将结果返回给请求地址的 activity。我们需要两个分别代表成功和失败的数字常量来报告地理编码过程的结果。定义一个 `Constants` 类来包含这些值，如下所示：

```
public final class Constants {
    public static final int SUCCESS_RESULT = 0;
    public static final int FAILURE_RESULT = 1;
    public static final String PACKAGE_NAME =
        "com.google.android.gms.location.sample.locationaddress";
    public static final String RECEIVER = PACKAGE_NAME + ".RECEIVER";
    public static final String RESULT_DATA_KEY = PACKAGE_NAME +
        ".RESULT_DATA_KEY";
    public static final String LOCATION_DATA_EXTRA = PACKAGE_NAME +
        ".LOCATION_DATA_EXTRA";
}
```

为了获取与地理位置相对应的街道地址，调用 getFromLocation()，传入位置对象的经度和纬度，以及我们想要返回的地址的最大数量。在这种情况下，我们只需要一个地址。geocoder 返回一个地址数组。如果没有找到匹配指定位置的地址，那么它会返回空的列表。如果没有可用的后台地理编码服务，geocoder 会返回 null。

如下面代码介绍来检查下述这些错误。如果出现错误，就将相应的错误信息传给变量 `errorMessage`，从而将错误信息发送给发出请求的 activity：

- **No location data provided** - Intent 的附加数据没有包含反向地理编码需要用到的 Location 对象。
- **Invalid latitude or longitude used** - Location 对象提供的纬度和/或者经度无效。
- **No geocoder available** - 由于网络错误或者 IO 异常，导致后台地理编码服务不可用。
- **Sorry, no address found** - geocoder 找不到指定纬度/经度对应的地址。

使用 Address 类中的 getAddressLine()) 方法来获得地址对象的个别行。然后将这些行加入一个地址 fragment 列表当中。其中，这个地址 fragment 列表准备好返回到发出地址请求的 activity。

为了将结果返回给发出地址请求的 activity，需要调用 `deliverResultToReceiver()` 方法（定义于下面的把地址返回给请求端）。结果由之前提到的成功/失败数字代码和一个字符串组成。在反向地理编码成功的情况下，这个字符串包含着地址。在失败的情况下，这个字符串

包含错误的信息。如下所示：

```
@Override
protected void onHandleIntent(Intent intent) {
    String errorMessage = "";

    // Get the location passed to this service through an extra.
    Location location = intent.getParcelableExtra(
            Constants.LOCATION_DATA_EXTRA);

    ...

    List<Address> addresses = null;

    try {
        addresses = geocoder.getFromLocation(
                location.getLatitude(),
                location.getLongitude(),
                // In this sample, get just a single address.
                1);
    } catch (IOException ioException) {
        // Catch network or other I/O problems.
        errorMessage = getString(R.string.service_not_available);
        Log.e(TAG, errorMessage, ioException);
    } catch (IllegalArgumentException illegalArgumentException) {
        // Catch invalid latitude or longitude values.
        errorMessage = getString(R.string.invalid_lat_long_used);
        Log.e(TAG, errorMessage + ". " +
                "Latitude = " + location.getLatitude() +
                ", Longitude = " +
                location.getLongitude(), illegalArgumentException);
    }

    // Handle case where no address was found.
    if (addresses == null || addresses.size()  == 0) {
        if (errorMessage.isEmpty()) {
            errorMessage = getString(R.string.no_address_found);
            Log.e(TAG, errorMessage);
        }
        deliverResultToReceiver(Constants.FAILURE_RESULT, errorMessage);
    } else {
        Address address = addresses.get(0);
        ArrayList<String> addressFragments = new ArrayList<String>();

        // Fetch the address lines using getAddressLine,
        // join them, and send them to the thread.
        for(int i = 0; i < address.getMaxAddressLineIndex(); i++) {
            addressFragments.add(address.getAddressLine(i));
        }
        Log.i(TAG, getString(R.string.address_found));
        deliverResultToReceiver(Constants.SUCCESS_RESULT,
                TextUtils.join(System.getProperty("line.separator"),
```

```
                          addressFragments));
    }
}
```

把地址返回给请求端

Intent 服务最后要做的事情是将地址返回给启动服务的 activity 里的 ResultReceiver。这个 ResultReceiver 类允许我们发送一个带有结果的数字代码和一个包含结果数据的消息。这个数字代码说明了地理编码请求是成功还是失败。在反向地理编码成功的情况下，这个消息包含着地址。在失败的情况下，这个消息包含一些描述失败原因的文本。

我们已经可以从 geocoder 取得地址，捕获到可能出现的错误，调用 `deliverResultToReceiver()` 方法。现在我们需要定义 `deliverResultToReceiver()` 方法来将结果代码和消息包发送给结果接收端。

对于结果代码，使用已经传给 `deliverResultToReceiver()` 方法的 `resultCode` 参数的值。对于消息包的结构，连接 `Constants` 类的 `RESULT_DATA_KEY` 常量（定义与获取街道地址数据）和传给 `deliverResultToReceiver()` 方法的 `message` 参数的值。如下所示：

```
public class FetchAddressIntentService extends IntentService {
    protected ResultReceiver mReceiver;
    ...
    private void deliverResultToReceiver(int resultCode, String message) {
        Bundle bundle = new Bundle();
        bundle.putString(Constants.RESULT_DATA_KEY, message);
        mReceiver.send(resultCode, bundle);
    }
}
```

启动 Intent 服务

上节课定义的 intent 服务在后台运行，同时，该服务负责提取与指定地理位置相对应的地址。当我们启动服务，Android 框架会实例化并启动服务（如果该服务没有运行），并且如果需要的话，创建一个进程。如果服务正在运行，那么让它保持运行状态。因为服务继承于 IntentService，所以当所有 intent 都被处理完之后，该服务会自动停止。

在我们应用的主 activity 中启动服务，并且创建一个 Intent 来把数据传给服务。我们需要创建一个显式的 intent，这是因为我们只想我们的服务响应该 intent。详细请见 Intent Types。

为了创建一个显式的 intent，需要为服务指定要用到的类名：`FetchAddressIntentService.class`。在 intent 附加数据中传入两个信息：

- 一个用于处理地址查找结果的 ResultReceiver。
- 一个包含想要转换成地址的纬度和经度的 Location 对象。

下面的代码介绍了如何启动 intent 服务：

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {

    protected Location mLastLocation;
    private AddressResultReceiver mResultReceiver;
    ...

    protected void startIntentService() {
        Intent intent = new Intent(this, FetchAddressIntentService.class);
        intent.putExtra(Constants.RECEIVER, mResultReceiver);
        intent.putExtra(Constants.LOCATION_DATA_EXTRA, mLastLocation);
        startService(intent);
    }
}
```

当用户请求查找地理地址时，调用上述的 `startIntentService()` 方法。例如，用户可能会在我们应用的 UI 上面点击提取地址按钮。在启动 intent 服务之前，我们需要检查是否已经连接到 Google Play services。下面的代码片段介绍在一个按钮 handler 中调用 `startIntentService()` 方法。

```
public void fetchAddressButtonHandler(View view) {
    // Only start the service to fetch the address if GoogleApiClient is
    // connected.
    if (mGoogleApiClient.isConnected() && mLastLocation != null) {
        startIntentService();
    }
    // If GoogleApiClient isn't connected, process the user's request by
    // setting mAddressRequested to true. Later, when GoogleApiClient connects,
    // launch the service to fetch the address. As far as the user is
    // concerned, pressing the Fetch Address button
    // immediately kicks off the process of getting the address.
    mAddressRequested = true;
    updateUIWidgets();
}
```

如果用户点击了应用 UI 上面的提取地址按钮，那么我们必须在 Google Play services 连接稳定之后启动 intent 服务。下面的代码片段介绍了调用 Google API Client 提供的 onConnected()) 回调函数中的 `startIntentService()` 方法。

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        // Gets the best and most recent location currently available,
        // which may be null in rare cases when a location is not available.
        mLastLocation = LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);

        if (mLastLocation != null) {
            // Determine whether a Geocoder is available.
            if (!Geocoder.isPresent()) {
                Toast.makeText(this, R.string.no_geocoder_available,
                        Toast.LENGTH_LONG).show();
                return;
            }

            if (mAddressRequested) {
                startIntentService();
            }
        }
    }
}
```

获取地理编码结果

Intent 服务已经处理完地理编码请求，并用 ResultReceiver 将结果返回给发出请求的 activity。在发出请求的 activity 里，定义一个继承于 ResultReceiver 的 `AddressResultReceiver`，用于处理在 `FetchAddressIntentService` 中的响应。

结果包含一个数字代码（`resultCode`）和一个包含结果数据（`resultData`）的消息。如果反向地理编码成功的话，`resultData` 会包含地址。如果失败，`resultData` 包含描述失败原因的文本。关于错误信息更详细的内容，请见把地址返回给请求端

重写 onReceiveResult() 方法来处理发送给接收端的结果，如下所示：

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    class AddressResultReceiver extends ResultReceiver {
        public AddressResultReceiver(Handler handler) {
            super(handler);
        }

        @Override
        protected void onReceiveResult(int resultCode, Bundle resultData) {

            // Display the address string
            // or an error message sent from the intent service.
            mAddressOutput = resultData.getString(Constants.RESULT_DATA_KEY);
            displayAddressOutput();

            // Show a toast message if an address was found.
            if (resultCode == Constants.SUCCESS_RESULT) {
                showToast(getString(R.string.address_found));
            }

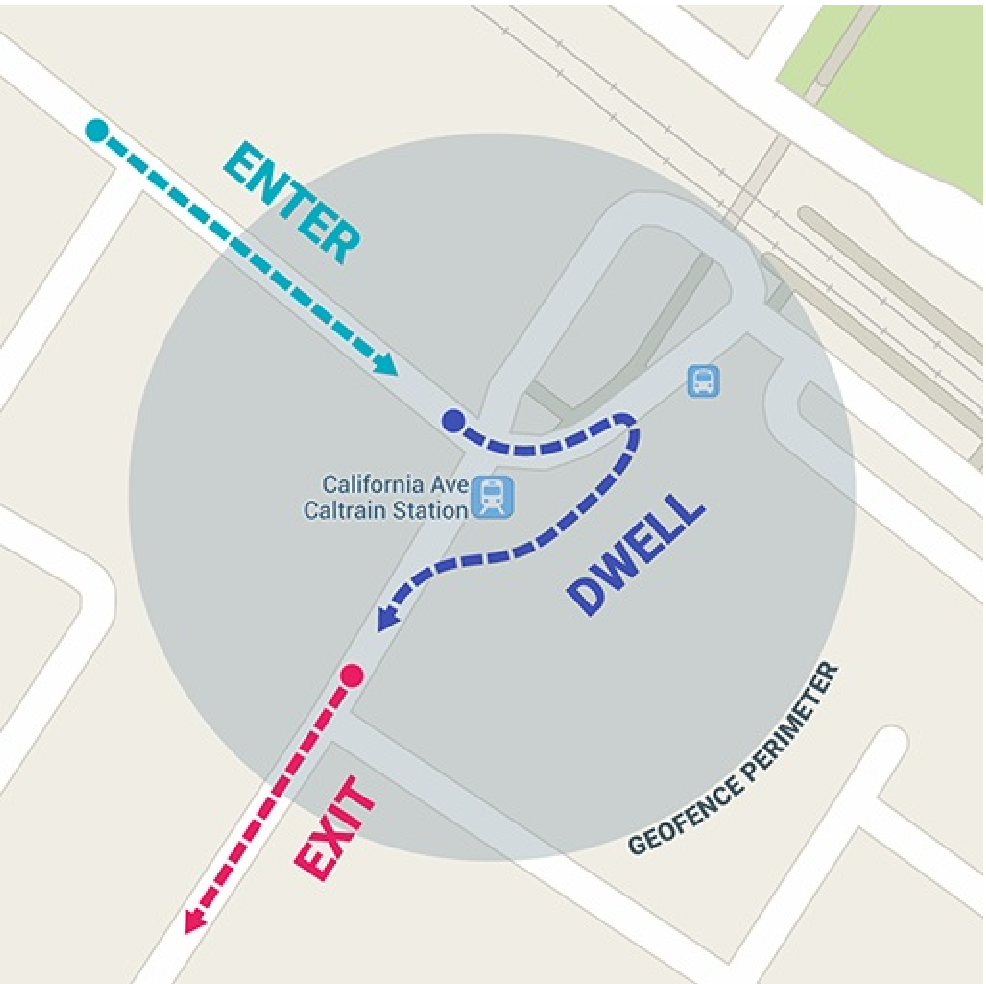
        }
    }
}
```

创建和监视地理围栏

> 编写:penkzhou - 原文:http://developer.android.com/training/location/geofencing.html

地理围栏将用户当前位置感知和附件地点特征感知相结合。为了标示一个感兴趣的位置，我们需要指定这个位置的经纬度。为了调整位置的邻近度，需要添加一个半径。经纬度和半径定义一个地理围栏，即在感兴趣的位置创建一个圆形区域或者围栏。

我们可以有多个活动的地理围栏（限制是一个设备用户100个）。对于每个地理围栏，我们可以让 Location Services 发出进入和离开事件，或者我们可以在触发一个事件之前，指定在某个地理围栏区域等待一段时间或者停留。通过指定一个以毫秒为单位的截止时间，我们可以限制任何一个地理围栏的持续时间。当地理围栏失效后，Location Services 会自动删除这个地理围栏。



这节课介绍如何添加和删除地理围栏，和用 IntentService 监听地理位置变化。

设置地理围栏监视

请求地理围栏监视的第一步就是设置必要的权限。在使用地理围栏时，我们必须设置 ACCESS_FINE_LOCATION 权限。在应用的 manifest 文件中添加如下子节点即可：

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

如果想要用 IntentService 监听地理位置变化，那么还需要添加一个节点来指定服务名字。这个节点必须是 的子节点：

```
<application
   android:allowBackup="true">
   ...
   <service android:name=".GeofenceTransitionsIntentService"/>
<application/>
```

为了访问位置 API，我们需要创建一个 Google Play services API client 的实例。想要学习如何连接 client，请见连接Google Play Services。

创建和添加地理围栏

我们的应用需要用位置 API 的 builder 类来创建地理围栏，用 convenience 类来添加地理围栏。另外，我们可以定义一个 PendingIntent（将在这节课介绍）来处理当地理位置发生迁移时，Location Services 发出的 intent。

创建地理围栏对象

首先，用 Geofence.Builder 创建一个地理围栏，设置想要的半径，持续时间，和地理围栏迁移的类型。例如，填充一个叫做 `mGeofenceList` 的 list 对象：

```
mGeofenceList.add(new Geofence.Builder()
    // Set the request ID of the geofence. This is a string to identify this
    // geofence.
    .setRequestId(entry.getKey())

    .setCircularRegion(
            entry.getValue().latitude,
            entry.getValue().longitude,
            Constants.GEOFENCE_RADIUS_IN_METERS
    )
    .setExpirationDuration(Constants.GEOFENCE_EXPIRATION_IN_MILLISECONDS)
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
            Geofence.GEOFENCE_TRANSITION_EXIT)
    .build());
```

这个例子从一个固定的文件中获取数据。在实际情况下，应用可能会根据用户的位置动态地创建地理围栏。

指定地理围栏和初始化触发器

下面的代码用到 GeofencingRequest 类。该类嵌套了 GeofencingRequestBuilder 类来需要监视的地理围栏和设置如何触发地理围栏事件：

```
private GeofencingRequest getGeofencingRequest() {
    GeofencingRequest.Builder builder = new GeofencingRequest.Builder();
    builder.setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER);
    builder.addGeofences(mGeofenceList);
    return builder.build();
}
```

这个例子介绍了两个地理围栏触发器。当设备进入一个地理围栏时，GEOFENCE_TRANSITION_ENTER 转移会触发。当设备离开一个地理围栏时，GEOFENCE_TRANSITION_EXIT 转移会触发。如果设备已经在地理围栏里面，那么指定 INITIAL_TRIGGER_ENTER 来通知位置服务触发 GEOFENCE_TRANSITION_ENTER。

在很多情况下，使用 INITIAL_TRIGGER_DWELL 可能会更好。仅仅当由于到达地理围栏中已定义好的持续时间，而导致用户停止时，INITIAL_TRIGGER_DWELL 才会触发事件。这个方法可以减少当设备短暂地进入和离开地理围栏时，由大量的通知造成的“垃圾警告信息”。另一种获取最好的地理围栏结果的策略是设置最小半径为100米。这有助于估计典型的 Wifi 网络的位置精确度，也有利于降低设备的功耗。

为地理围栏转移定义Intent

从 Location Services 发送来的Intent能够触发各种应用内的动作，但是不能用它来打开一个 Activity 或者 Fragment，这是因为应用内的组件只能在响应用户动作时才可见。大多数情况下，处理这一类 Intent 最好使用 IntentService。一个 IntentService 可以推送一个通知，可以进行长时间的后台作业，可以将 intent 发送给其他的 services ，还可以发送一个广播 intent。下面的代码展示了如何定义一个 PendingIntent 来启动一个 IntentService:

```
public class MainActivity extends FragmentActivity {
    ...
    private PendingIntent getGeofencePendingIntent() {
        // Reuse the PendingIntent if we already have it.
        if (mGeofencePendingIntent != null) {
            return mGeofencePendingIntent;
        }
        Intent intent = new Intent(this, GeofenceTransitionsIntentService.class);
        // We use FLAG_UPDATE_CURRENT so that we get the same pending intent back when
        // calling addGeofences() and removeGeofences().
        return PendingIntent.getService(this, 0, intent, PendingIntent.
                FLAG_UPDATE_CURRENT);
    }
```

添加地理围栏

使用 GeoencingApi.addGeofences() 方法来添加地理围栏。为该方法提供 Google API client，GeofencingRequest 对象和 PendingIntent。下面的代码，在 onResult()) 中处理结果，假设主 activity 实现 ResultCallback。

```
public class MainActivity extends FragmentActivity {
    ...
    LocationServices.GeofencingApi.addGeofences(
                mGoogleApiClient,
                getGeofencingRequest(),
                getGeofencePendingIntent()
        ).setResultCallback(this);
```

处理地理围栏转移

当 Location Services 探测到用户进入或者离开一个地理围栏，它会发送一个包含在 PendingIntent 的 Intent，这个 PendingIntent 就是在添加地理围栏时被我们包括在请求当中。这个 Intent 被一个类似 `GeofenceTransitionsIntentService` 的 service 接收，这个 service 从 intent 得到地理围栏事件，决定地理围栏转移的类型，和决定触发哪个已定义的地理围栏。然后它会发出一个通知。

下面的代码介绍了如何定义一个 IntentService。这个 IntentService 在地理围栏转移出现时，会推送一个通知。当用户点击这个通知，那么应用的主 activity 会出现：

```java
public class GeofenceTransitionsIntentService extends IntentService {
   ...
    protected void onHandleIntent(Intent intent) {
        GeofencingEvent geofencingEvent = GeofencingEvent.fromIntent(intent);
        if (geofencingEvent.hasError()) {
            String errorMessage = GeofenceErrorMessages.getErrorString(this,
                    geofencingEvent.getErrorCode());
            Log.e(TAG, errorMessage);
            return;
        }

        // Get the transition type.
        int geofenceTransition = geofencingEvent.getGeofenceTransition();

        // Test that the reported transition was of interest.
        if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER ||
                geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {

            // Get the geofences that were triggered. A single event can trigger
            // multiple geofences.
            List triggeringGeofences = geofencingEvent.getTriggeringGeofences();

            // Get the transition details as a String.
            String geofenceTransitionDetails = getGeofenceTransitionDetails(
                    this,
                    geofenceTransition,
                    triggeringGeofences
            );

            // Send notification and log the transition details.
            sendNotification(geofenceTransitionDetails);
            Log.i(TAG, geofenceTransitionDetails);
        } else {
            // Log the error.
            Log.e(TAG, getString(R.string.geofence_transition_invalid_type,
                    geofenceTransition));
        }
    }
```

在通过 PendingIntent 检测转移事件之后，这个 IntentService 获取地理围栏转移类型和测试一个事件是不是应用用来触发通知的 —— 要么是 GEOFENCE_TRANSITION_ENTER，要么是 GEOFENCE_TRANSITION_EXIT。然后，这个 service 会发出一个通知并且记录转移的详细信息。

停止地理围栏监视

当不再需要监视地理围栏或者想要节省设备的电池电量和 CPU 周期时，需要停止地理围栏监视。我们可以在用于添加和删除地理围栏的主 activity 里停止地理围栏监视；删除地理围栏会导致它马上停止。API 要么通过 request IDs，要么通过删除与指定 PendingIntent 相关的地理围栏来删除地理围栏。

下面的代码通过 PendingIntent 删除地理围栏，当设备进入或者离开之前已经添加的地理围栏时，停止所有通知：

```
LocationServices.GeofencingApi.removeGeofences(
            mGoogleApiClient,
            // This is the same pending intent that was used in addGeofences().
            getGeofencePendingIntent()
    ).setResultCallback(this); // Result processed in onResult().
}
```

你可以将地理围栏同其他位置感知的特性结合起来，比如周期性的位置更新。像要了解更多的信息，请看本章的其它课程。

Android可穿戴应用

> 编写:kesenhoo - 原文: http://developer.android.com/training/building-wearables.html

这些课程将教我们如何在手持应用上构建notification，并且使得这些notification能够自动同步到可穿戴设备上。同样也会教我们如何创建直接运行在可穿戴设备上的应用。

> **Note**：关于这几节课用到的API的详细信息，请见Wear API reference documentation。

赋予Notification可穿戴的特性

学习如何构建运行在手持设备的上得notification并且使得他们能够同步到可穿戴上设备时有良好的体验。

创建可穿戴应用

学习如何构建直接运行在可穿戴设备上的应用。

创建自定义的UI

学习如何为可穿戴应用创建自定义的界面。

发送与同步数据

学习如何在手持设备与可穿戴设备之间同步数据。

创建表盘

学习如何创建表盘。

检测位置

学习如何在Android穿戴设备上检测位置数据。

为Notification赋加可穿戴特性

> 编写:wangyachen - 原文:
> http://developer.android.com/training/wearables/notifications/index.html

当一部Android手持设备（手机或平板）与Android可穿戴设备连接后，手持设备能够自动与可穿戴设备共享Notification。在可穿戴设备上，每个Notification都是以一张新卡片的形式出现在context stream中。

与此同时，为了给予用户以最佳的体验，开发者应当为自己创建的Notification增加一些具备可穿戴特性的功能。下面的课程将指导我们如何实现同时支持手持设备和可穿戴设备的Notification。

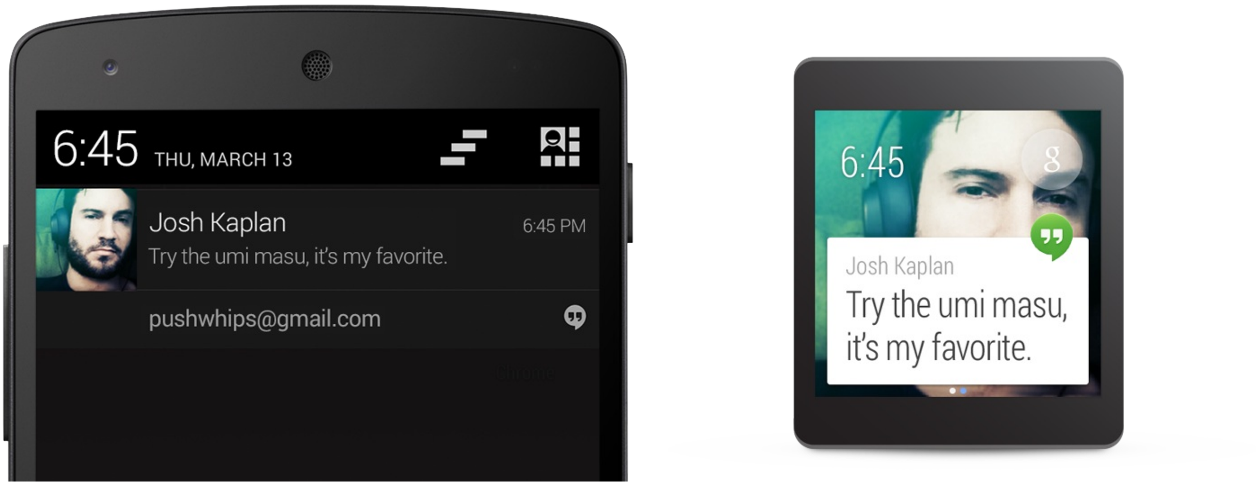


Figure 1. 同时展示在手持设备和可穿戴设备的Notification

Lessons

创建Notification

学习如何应用Android support library创建具备可穿戴特性的Notification。

在Notification中接收语音输入

学习在可穿戴式设备上的Notification添加一个action以接收来自用户的语音输入，并且将录入的消息传递给手持设备应用。

为Notification添加页面

学习如何为Notification创建附加的页面，使得用户在向左滑动时能看到更多的信息。

将Notification放成一叠

学习如何将我们应用中所有相似的notification放在一个堆叠中，使得在不将多个卡片添加到卡片流的情况下，允许用户能够独立地查看每一个Notification。

为可穿戴设备创建Notification

> 编写:wangyachen - 原文:
> http://developer.android.com/training/wearables/notifications/creating.html

使用 NotificationCompat.Builder 来创建可以发送给可穿戴设备的手持设备Notification。当我们使用这个类创建Notification之后，无论Notification出现在手持式设备上还是可穿戴设备上，系统都会把Notification正确地显示出来。

> **Note**：使用 RemoteViews 的Notification会剥除自定义的 layout，并且可穿戴设备上只显示文本和图标。但是，通过创建一个运行在可穿戴设备上的应用，开发者能够使用自定义的卡片布局创建自定义Notifications。

Import必要的类

为了引入必要的包，在我们的 `build.gradle` 文件中加入如下内容：

```
compile "com.android.support:support-v4:20.0.+"
```

现在我们的项目能够访问关键的包，接下来从support library中引入必要的类：

```
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.NotificationCompat.WearableExtender;
```

通过Notification Builder创建Notification

v4 support library能够让开发者使用最新的特性去创建 Notification，诸如action 按钮和大的图标，而且兼容Android1.6（API level4）及以上的版本。

为了通过support library创建一个Notification，我们需要创建一个 NotificationCompat.Builder 的实例，然后通过将该实例传给 notify()) 来发出 Notification。例如：

```
int notificationId = 001;
// Build intent for notification content
Intent viewIntent = new Intent(this, ViewEventActivity.class);
viewIntent.putExtra(EXTRA_EVENT_ID, eventId);
PendingIntent viewPendingIntent =
        PendingIntent.getActivity(this, 0, viewIntent, 0);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent);

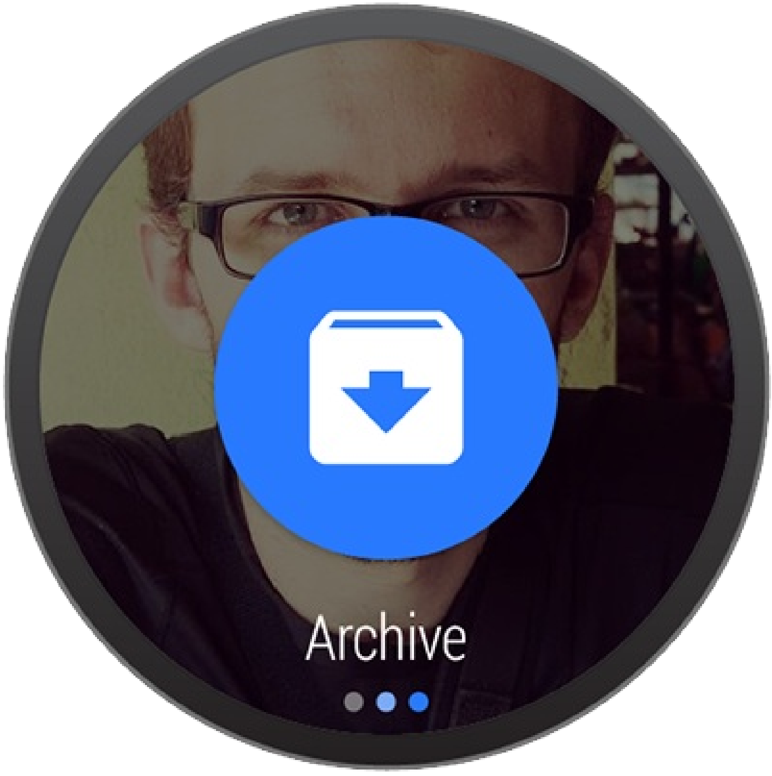
// Get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);

// Build the notification and issues it with notification manager.
notificationManager.notify(notificationId, notificationBuilder.build());
```

当该Notification出现在手持设备上时，用户能够通过触摸Notification来触发之前通过setContentIntent()设置的PendingIntent。当该Notification出现在可穿戴设备上时，用户能够通过向左滑动该Notification显示**Open**的action，点击这个action能够激活手持设备上的Intent。

添加Action按钮

除了通过 setContentIntent()) 定义的主要内容action之外，我们还可以通过传递一个PendingIntent 给 addAction()) 来添加其它action。



例如，下面的代码展示了创建一个同之前相仿的Notification，只不过添加了一个在地图上查看事件位置的action。

```
// Build an intent for an action to view a map
Intent mapIntent = new Intent(Intent.ACTION_VIEW);
Uri geoUri = Uri.parse("geo:0,0?q=" + Uri.encode(location));
mapIntent.setData(geoUri);
PendingIntent mapPendingIntent =
        PendingIntent.getActivity(this, 0, mapIntent, 0);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent)
        .addAction(R.drawable.ic_map,
                getString(R.string.map), mapPendingIntent);
```

在手持设备上，action表现为在Notification上附加的一个额外按钮。而在可穿戴设备上，action表现为Notification左滑后出现的大按钮。当用户点击action时，能够触发手持设备上对应的intent。

> **Tip**：如果我们的Notification包含了一个"回复"的action(例如短信类app)，我们可以通过支持直接从Android可穿戴设备返回的语音输入，来加强该功能的体验。更多信息，详见在Notification中接收语音输入。

可穿戴式独有的 Actions

如果开发者想要可穿戴式设备上的action与手持式设备不一样的话，可以使用WearableExtender.addAction())，一旦我们通过这种方式添加了action，可穿戴式设备便不会显示任何其他通过 NotificationCompat.Builder.addAction()) 添加的action。这是因为，只有通过 WearableExtender.addAction()) 添加的action才能只在可穿戴设备上显示且不在手持式设备上显示。

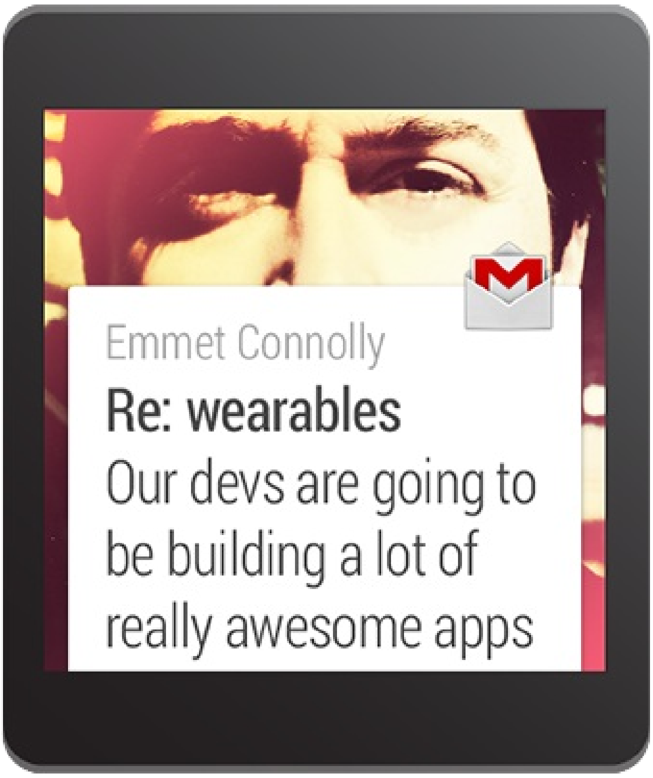
```
// Create an intent for the reply action
Intent actionIntent = new Intent(this, ActionActivity.class);
PendingIntent actionPendingIntent =
        PendingIntent.getActivity(this, 0, actionIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);

// Create the action
NotificationCompat.Action action =
        new NotificationCompat.Action.Builder(R.drawable.ic_action,
                getString(R.string.label, actionPendingIntent))
                .build();

// Build the notification and add the action via WearableExtender
Notification notification =
        new NotificationCompat.Builder(mContext)
                .setSmallIcon(R.drawable.ic_message)
                .setContentTitle(getString(R.string.title))
                .setContentText(getString(R.string.content))
                .extend(new WearableExtender().addAction(action))
                .build();
```

添加一个Big View

开发者可以在Notification中通过添加某种"big view"风格来插入扩展文本。在手持式设备上，用户能够通过展开Notification看见big view的内容。在可穿戴式设备上，big view内容是默认可见的。



可以通过 NotificationCompat.Builder 对象调用 setStyle())，并设置参数为 BigTextStyle 或 InboxStyle 的实例，从而将扩展内容添加到 Notification 中。

比如，下面的代码为事件 Notification 添加了一个 NotificationCompat.BigTextStyle 的实例，目的是为了包含完整的事件描述(这能够包含比 setContentText()) 提供的空间所能容纳的字数更多的文字)。

```
// Specify the 'big view' content to display the long
// event description that may not fit the normal content text.
BigTextStyle bigStyle = new NotificationCompat.BigTextStyle();
bigStyle.bigText(eventDescription);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setLargeIcon(BitmapFractory.decodeResource(
                getResources(), R.drawable.notif_background))
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent)
        .addAction(R.drawable.ic_map,
                getString(R.string.map), mapPendingIntent)
        .setStyle(bigStyle);
```

要注意的是，开发者可以通过 setLargeIcon()) 方法为任何 Notification 添加一个大图标。但是，这些图标在可穿戴设备上会显示成大的背景图片，并且由于这些图标会被放大以适应可穿戴设备的屏幕，导致这些图标显示的效果不好。想要为 Notification 添加一个可穿戴设备适

用的背景图片，请看下面一小节为 Notification 添加可穿戴式特性。更多关于大图片在 Notification 上的设计，详见 Design Principles of Android Wear。

为Notification添加可穿戴式特性

如果我们需要为 Notification 添加一些可穿戴式的特性设置，比如制定额外的内容页，或者让用户通过语音输入一些文字，那么我们可以使用 NotificationCompat.WearableExtender 来制定这些设置。为了适用这个 API，我们需要：

1. 创建一个 WearableExtender 的实例，为 Notification 设置可穿戴设备独有的特性。
2. 创建一个 NotificationCompat.Builder 的实例，就像本课程先前所说的，设置需要的 Notification 属性。
3. 调用 Notification 上的 extend()) 并将 WearableExtender 传进该方法。这在 Notification 上应用了可穿戴设备的选项。
4. 调用 build()) 去构建一个 Notification。

例如，以下代码调用 setHintHideIcon()) 方法把应用的图标从 Notification 卡片上删掉。

```
// Create a WearableExtender to add functionality for wearables
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setHintHideIcon(true)
        .setBackground(mBitmap);

// Create a NotificationCompat.Builder to build a standard notification
// then extend it with the WearableExtender
Notification notif = new NotificationCompat.Builder(mContext)
        .setContentTitle("New mail from " + sender)
        .setContentText(subject)
        .setSmallIcon(R.drawable.new_mail)
        .extend(wearableExtender)
        .build();
```

setHintHideIcon()) 和 setBackground()) 这两个方法是 NotificationCompat.WearableExtender 可用的新 Noticication 特性的两个例子。

> **Note**：setBackground()) 中使用的位图在不滚动的背景下应该是 400x400 的分辨率，在支持视差滚动的背景下应该是 640x640。将这些位图放在 `res/drawable-nodpi` 目录下。将可穿戴 Notification 中使用的其它不是位图的资源放到 `res/drawable-hdpi` 目录，例如 setContentIcon()) 用到的那些资源。

如果开发者需要稍后去读取可穿戴特性的设置，可以使用设置相应的get方法，该例子通过调用 getHintHideIcon()) 去获取当前 Notification 是否隐藏了图标。

```
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender(notif);
boolean hintHideIcon = wearableExtender.getHintHideIcon();
```

传递 Notification

如果开发者想要传递自己的 Notification，请使用 NotificationManagerCompat 的API代替 NotificationManager：

```
// Get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(mContext);

// Issue the notification with notification manager.
notificationManager.notify(notificationId, notif);
```

如果开发者使用了framework中的 NotificationManager ，那么 NotificationCompat.WearableExtender 中的一些特性就会失效，所以，请确保使用 NotificationManagerCompat。

下一课：在 Notifcation 中接收语音输入

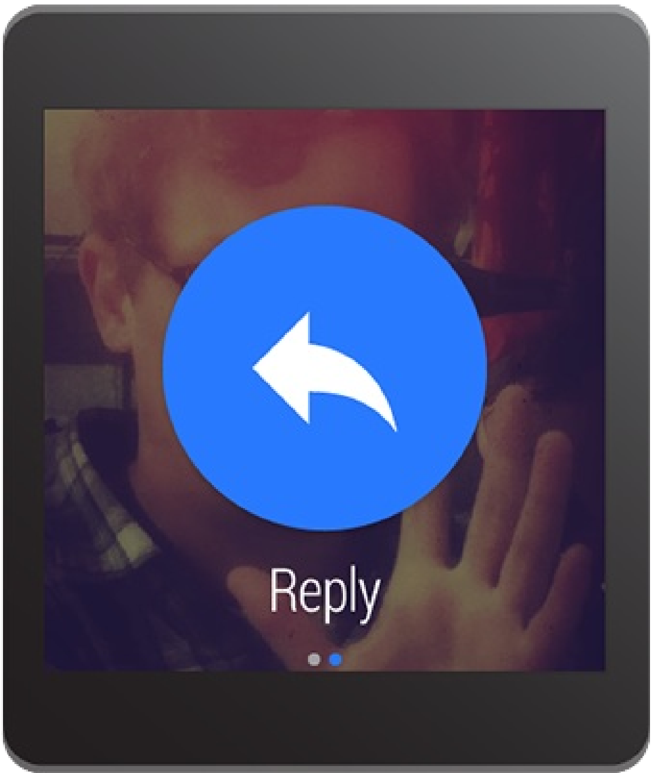
在 Notifcation 中接收语音输入

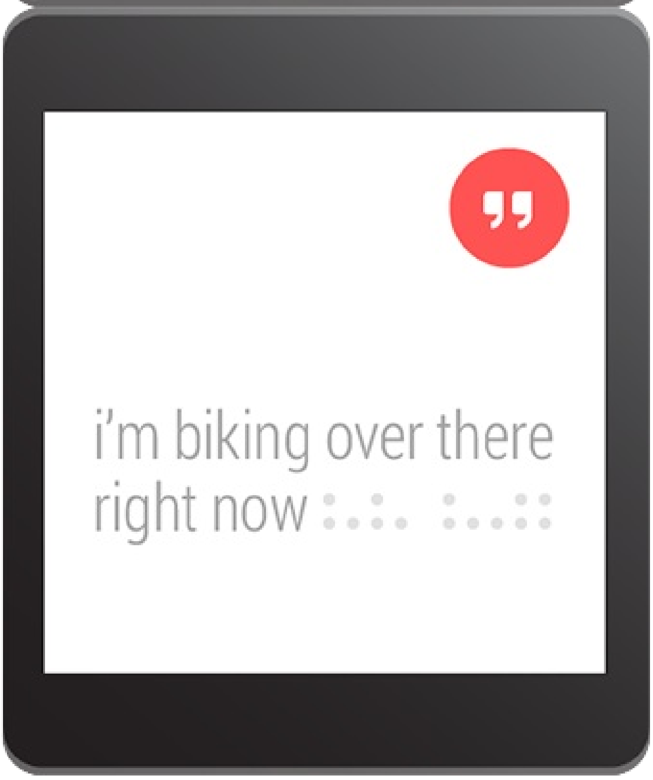
> 编写:wangyachen - 原文:http://developer.android.com/training/wearables/notifications/voice-input.html

如果手持式设备上的Notification包含了一个输入文本的action，比如回复邮件，那么这个action正常情况下应该会调起一个activity让用户进行输入。但是，当这个action出现在可穿戴式设备上时，是没有键盘可以让用户进行输入的，所以开发者应该让用户指定一个反馈或者通过RemoteInput预先设定好文本信息。

当用户通过语音或者选择可见的消息进行回复时，系统会将文本的反馈信息与开发者指定的Notification中的action中的Intent进行绑定，并且将该intent发送给手持设备中的app。

> **Note**：Android模拟器并不支持语音输入。如果使用可穿戴式设备的模拟器的话，可以打开AVD设置中的**Hardware keyboard present**，实现用打字代替语音。





定义语音输入

为了创建一个支持语音输入的action，需要创建一个RemoteInput.Builder的实例，将其加到Notification的action中。这个类的构造函数接受一个String类型的参数，系统用这个参数作为语音输入的key，后面我们会用这个key来取得在手持设备中输入的文本。

举个例子，下面展示了如何创建一个RemoteInput对象，其中，该提供了一个用于提示语音输入的自定义label。

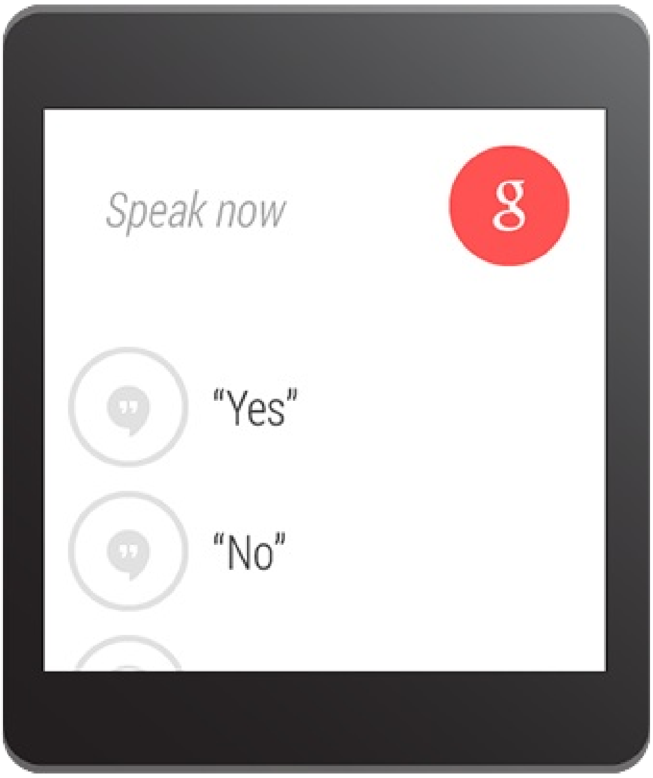
```
// Key for the string that's delivered in the action's intent
private static final String EXTRA_VOICE_REPLY = "extra_voice_reply";

String replyLabel = getResources().getString(R.string.reply_label);

RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
        .setLabel(replyLabel)
        .build();
```

添加预先设定的文本反馈

除了要打开语音输入支持之外，开发者还可以提供多达5条的文本反馈，这样用户可以直接选择实现快速回复。该功能可通过调用setChoices())并传递一个String数组实现。



举个例子，可以用resource数组的方式定义这些反馈：

`res/values/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="reply_choices">
        <item>Yes</item>
        <item>No</item>
        <item>Maybe</item>
    </string-array>
</resources>
```

然后，填充 String 数组，并将其添加到RemoteInput中：

```
public static final EXTRA_VOICE_REPLY = "extra_voice_reply";
...
String replyLabel = getResources().getString(R.string.reply_label);
String[] replyChoices = getResources().getStringArray(R.array.reply_choices);

RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
        .setLabel(replyLabel)
        .setChoices(replyChoices)
        .build();
```

添加语音输入作为Notification的action

为了实现设置语音输入，可以把RemoteInput对象通过addRemoteInput()设置到一个action中。然后我们可以将这个action应用到Notification中，例如：

```
// Create an intent for the reply action
Intent replyIntent = new Intent(this, ReplyActivity.class);
PendingIntent replyPendingIntent =
        PendingIntent.getActivity(this, 0, replyIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);

// Create the reply action and add the remote input
NotificationCompat.Action action =
        new NotificationCompat.Action.Builder(R.drawable.ic_reply_icon,
                getString(R.string.label, replyPendingIntent))
                .addRemoteInput(remoteInput)
                .build();

// Build the notification and add the action via WearableExtender
Notification notification =
        new NotificationCompat.Builder(mContext)
                .setSmallIcon(R.drawable.ic_message)
                .setContentTitle(getString(R.string.title))
                .setContentText(getString(R.string.content))
                .extend(new WearableExtender().addAction(action))
                .build();

// Issue the notification
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(mContext);
notificationManager.notify(notificationId, notification);
```

当程序发出这个Notification的时候，用户在可穿戴设备上左滑便可以看到reply的按钮。

将语音输入转化为String

通过调用getResultsFromIntent())方法，将返回值放在"Reply"的action指定的intent中，开发者便可以在回复的action的intent中指定的activity里，接收到用户转录后的消息。该方法返回的是包含了文本反馈的Bundle。我们可以通过查询Bundle中的内容来获得这条反馈。

> **Note**：请不要使用Intent.getExtras())来获取语音输入的结果，因为语音输入的内容是存储在ClipData中的。getResultsFromIntent())提供了一条很方便的途径来接收字符数组类型的语音信息，并且不需要经过ClipData自身的调用。

下面的代码展示了一个接收intent，并且返回语音反馈信息的方法，该方法是依据之前例子中的 `EXTRA_VOICE_REPLY` 作为key进行检索：

```
/**
 * Obtain the intent that started this activity by calling
 * Activity.getIntent() and pass it into this method to
 * get the associated voice input string.
 */

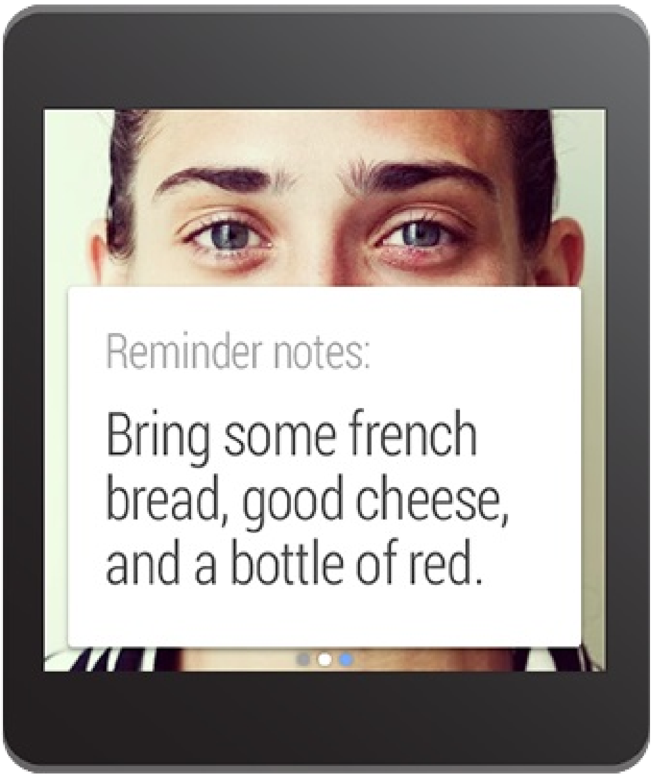
private CharSequence getMessageText(Intent intent) {
    Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
        if (remoteInput != null) {
            return remoteInput.getCharSequence(EXTRA_VOICE_REPLY);
        }
    }
    return null;
}
```

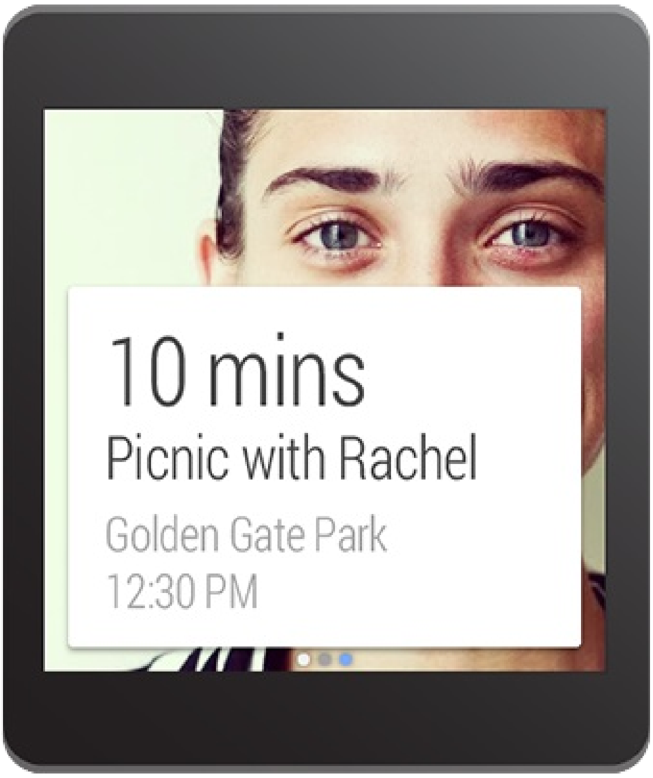
下一课：为Notification添加页面

为 Notification 添加页面

> 编写:wangyachen - 原文:http://developer.android.com/training/wearables/notifications/pages.html

当开发者想要在不需要用户在他们的手机上打开app的情况下，还可以允许表达更多的信息，那么开发者可以在可穿戴设备上的Notification中添加一个或多个的页面。添加的页面会马上出现在主 Notification 卡片的右边。





为了创建一个拥有多个页面的 Notification，开发者需要：

1. 通过NotificationCompat.Builder创建主Notification（首页），以开发者想要的方式使其出现在手持设备上。
2. 通过NotificationCompat.Builder为可穿戴设备添加更多的页面。
3. 通过addPage())方法将这些页面应用到主 Notification 中，或者通过addPages())将多个页面添加到一个Collection。

举个例子，以下代码为Notification添加了第二个页面：

```
// Create builder for the main notification
NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.new_message)
        .setContentTitle("Page 1")
        .setContentText("Short message")
        .setContentIntent(viewPendingIntent);

// Create a big text style for the second page
BigTextStyle secondPageStyle = new NotificationCompat.BigTextStyle();
secondPageStyle.setBigContentTitle("Page 2")
               .bigText("A lot of text...");

// Create second page notification
Notification secondPageNotification =
        new NotificationCompat.Builder(this)
        .setStyle(secondPageStyle)
        .build();

// Add second page with wearable extender and extend the main notification
Notification twoPageNotification =
        new WearableExtender()
                .addPage(secondPageNotification)
                .extend(notificationBuilder)
                .build();

// Issue the notification
notificationManager =
        NotificationManagerCompat.from(this);
notificationManager.notify(notificationId, twoPageNotification);
```

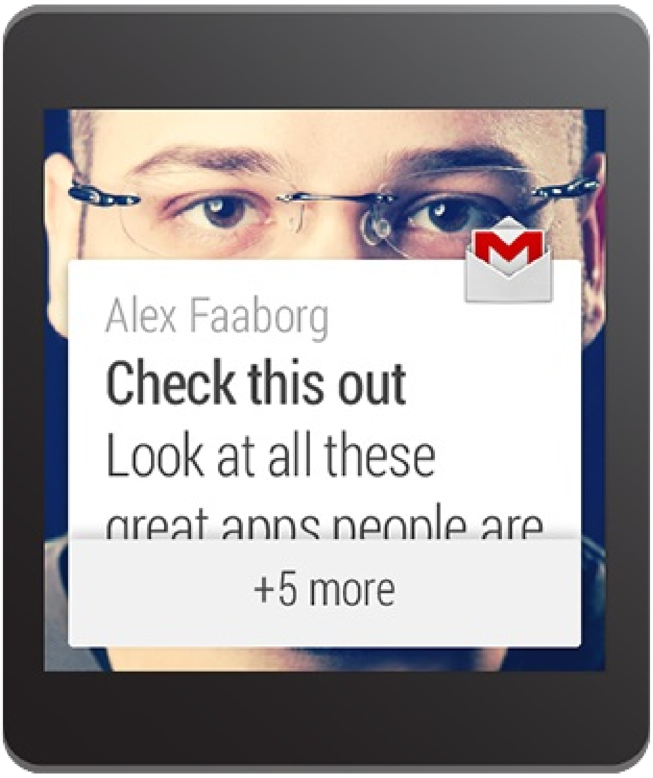
下一课：以Stack的方式显示Notifications

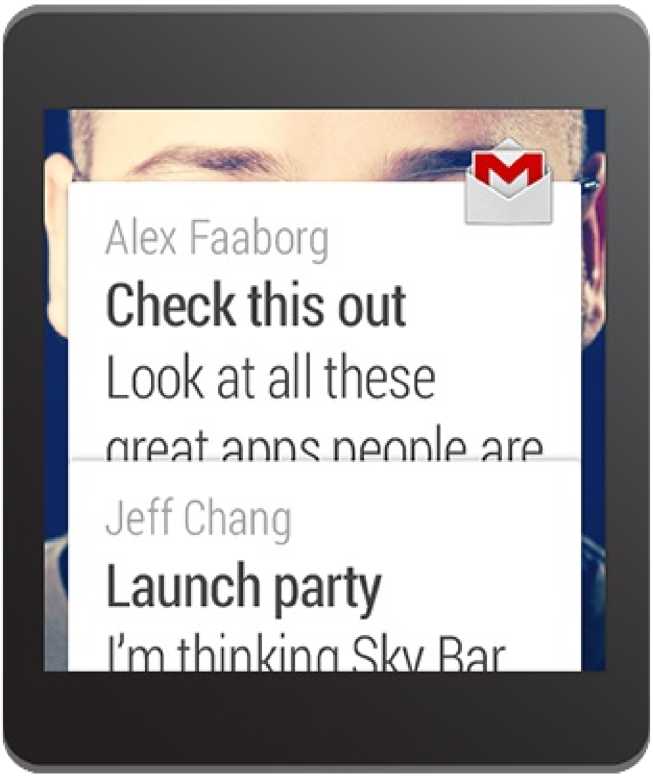
将 Notification 放成一叠

> 编写:wangyachen - 原文:
> http://developer.android.com/training/wearables/notifications/stacks.html

当为手持式设备创建Notification时，开发者应该将多个相似的Notification合并成一个概括式的Notification。例如，如果app创建了一系列接收短信的Notification，开发者不应该将多于一个Notification显示到可穿戴设备上——当接收到多于一条消息的时候，用一个Notification提供一个摘要，比如"2条新消息"。

尽管如此，一个概括式的Notification在可穿戴设备上并不是很有用处，因为用户不能在可穿戴设备上阅读每条消息的详细内容(他们必须在手持式设备上打开相应的app才能看到更多信息)。所以对可穿戴设备而言，开发者应该将所有的Notification都集中起来，放成一叠。这叠Notification以一张卡片的形式显示出来，用户可以将它展开，分别看到每个Notification的详细内容。通过新方法setGroup())能够实现该功能，同时，也能保持手持式设备上显示为一条概括式的Notification。





将每个Notification添加到一个群组中

为了创建一个stack，可以对每个想要放入该stack的Notification调用setGroup())，并且指定一个group key。然后调用notify())将其发送至可穿戴设备上。

```
final static String GROUP_KEY_EMAILS = "group_key_emails";

// Build the notification, setting the group appropriately
Notification notif = new NotificationCompat.Builder(mContext)
         .setContentTitle("New mail from " + sender1)
         .setContentText(subject1)
         .setSmallIcon(R.drawable.new_mail);
         .setGroup(GROUP_KEY_EMAILS)
         .build();

// Issue the notification
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);
notificationManager.notify(notificationId1, notif);
```

稍后，当开发者创建另一个Notification的时候，指定同样的group key。当在调用notify())的时候，这个Notification就会出现在之前那个Notification的同一个stack中，而非新建一张卡片。

```
Notification notif2 = new NotificationCompat.Builder(mContext)
         .setContentTitle("New mail from " + sender2)
         .setContentText(subject2)
         .setSmallIcon(R.drawable.new_mail);
         .setGroup(GROUP_KEY_EMAILS)
         .build();

notificationManager.notify(notificationId2, notif2);
```

在默认的情况下，Notification的排列顺序由开发者添加的先后顺序决定，最近的Notification会被放置在最顶端。你可以通过setSortKey())来修改Notification的排顺序。

添加概括式Notification

在手持设备上提供一个概括式的Notification是很重要的。因此除了要将每条单独的Notification放置在同一个stack group中，还需要添加一个概括式的Notification，并对其调用setGroupSummary())即可实现。

该Notification并不会出现在可穿戴设备上的stack中，只会出现在手持式设备上。




```
Bitmap largeIcon = BitmapFactory.decodeResource(getResources(),
        R.drawable.ic_large_icon);

// Create an InboxStyle notification
Notification summaryNotification = new NotificationCompat.Builder(mContext)
        .setContentTitle("2 new messages")
        .setSmallIcon(R.drawable.ic_small_icon)
        .setLargeIcon(largeIcon)
        .setStyle(new NotificationCompat.InboxStyle()
                .addLine("Alex Faaborg   Check this out")
                .addLine("Jeff Chang   Launch Party")
                .setBigContentTitle("2 new messages")
                .setSummaryText("johndoe@gmail.com"))
        .setGroup(GROUP_KEY_EMAILS)
        .setGroupSummary(true)
        .build();

notificationManager.notify(notificationId3, summaryNotification);
```

该Notification使用了NotificationCompat.InboxStyle，这个style能够让开发者很轻松地创建邮件或者短信app的Notifications。开发者可以对概括式Notification使用这个style，或者NotificationCompat中定义的其他style，或者不使用任何style也可以。

> **Tip**：如果想要和上面截图中一样的设计文本，请参考Styling with HTML markup和Styling with Spannables。

概括式Notification能够在不显示在可穿戴设备上的前提下做到影响可穿戴设备上的Notification。当开发者创建一个概括式Notification时，可以利用NotificationCompat.WearableExtender，调用setBackground()或者addAction()为可穿戴设备上的整个stack设置一个背景图片或者一个action。以下代码展示了如何为整个stack设置背景：

```
Bitmap background = BitmapFactory.decodeResource(getResources(),
        R.drawable.ic_background);

NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setBackground(background);

// Create an InboxStyle notification
Notification summaryNotificationWithBackground =
        new NotificationCompat.Builder(mContext)
        .setContentTitle("2 new messages")
        ...
        .extend(wearableExtender)
        .setGroup(GROUP_KEY_EMAILS)
        .setGroupSummary(true)
        .build();
```

下一课：创建可穿戴的应用

创建可穿戴的应用

> 编写:kesenhoo - 原文:http://developer.android.com/training/wearables/apps/index.html

可穿戴应用直接运行在穿戴设备上，应用可以直接访问例如传感器与GPU这样的硬件。这些应用和一般的Android应用的基础部分是一致的，只是在设计与可用性还有一些特殊功能上有比较大差异。手持设备与可穿戴设备上的应用主要有下面的一些差异：

- 系统会强制执行超时机制。如果我们显示了一个Activity，用户并没有进行操作，设备会进入睡眠状态。当设备唤醒时，穿戴设备会显示主界面而不是刚才的activity。如果我们想要持续的显示一些东西，请使用notification来替代。
- 相比起手持设备的应用，可穿戴应用的界面相对更小，功能也相对更少。他仅仅包含了那些对于可穿戴有意义的功能，这些功能通常是手持设备的一个子集。通常来说，我们应该尽可能的把运行操作搬到手持设备上，然后发送操作结果到可穿戴设备。
- 用户不会直接将应用下载到可穿戴设备上进行安装。相反，我们将可穿戴设备应用打包到手持设备应用里。当用户安装手持设备的应用时，系统会自动安装可穿戴应用。然而，为了开发便利，我们还是可以直接安装应用到可穿戴设备。
- 可穿戴应用可以使用大多数的标准Android APIs，除了下面的以外：
    - android.webkit
    - android.print
    - android.app.backup
    - android.appwidget
    - android.hardware.usb

  在使用某个API之前，我们可以通过执行hasSystemFeature()) 来判断可穿戴应用是否支持某个功能。

> **Note:** 我们推荐使用Android Studio来开发Android Wear的应用，因为它提供了建立工程，添加库依赖，打包程序等在ADT上没有的功能。下面的培训课程的前提是假设你已经在使用Android Studio了。

Lessons

- 创建并运行可穿戴应用(Creating and Running a Wearable App)

  学习如何创建一个包含了可穿戴与手持应用的Android Studio工程。学习如何在设备或者模拟器上运行应用。

- 创建自定义的布局(Creating Custom Layouts)

  学习如何为notification与activity创建并显示一个自定义的布局

- 添加语音功能(Adding Voice Capabilities)

  学习如何使用语音指令启动一个activity，学习如何启动系统语音识别应用来获取用户的语音输入。

- 打包可穿戴应用(Packaging Wearable Apps)

  学习如何把可穿戴应用打包到手持应用上。这使得系统能够在安装Google Play商店上的手持应用时自动安装可穿戴应用。

- 通过蓝牙进行调试(Debugging over Bluetooth)

  学习如何通过蓝牙而不是USB来调试可穿戴应用。

创建并运行可穿戴应用

> 编写:kesenhoo - 原文:http://developer.android.com/training/wearables/apps/creating.html

可穿戴应用可以直接运行在可穿戴的设备上。拥有访问类似传感器的硬件权限，还有操作activity，services等权限。

当我们想要将可穿戴设备应用发布到Google Play商店时，我们需要有该应用的配套手持设备应用。因为可穿戴设备不支持Google Play商店，所以当用户下载配套手持设备应用的时候，会自动安装可穿戴应用到可穿戴设备上。手持设备应用还可以用来处理一些繁重的任务、网络指令或者其它工作，和发送操作结果给可穿戴设备。

这节课会介绍如何安装一个设备或者模拟器，和如何创建一个包含了手持应用与可穿戴应用的工程。

升级 SDK

在开始建立可穿戴设备应用前，必须：

- 将**SDK**工具升级到**23.0.0**或者更高的版本

  升级后的SDK工具使我们可以建立和测试可穿戴应用。

- 将**SDK**升级到 **Android 4.4W.2(API 20)** 或者更高

  升级后的平台版本为可穿戴应用提供了新的 API。

想要了解如何升级SDK，请查看Get the latest SDK tools。

搭建**Android Wear**模拟器或者真机设备。

我们推荐在真机上进行开发，这样可以更好地评估用户体验。然而，模拟器可以使我们在不同类型的设备屏幕上进行模拟，这对测试来说非常有用。

搭建**Android Wear**虚拟设备

建立Android Wear虚拟设备需要下面几个步骤：

1. 点击**Tools > Android > AVD Manager**。
2. 点击**Create Virtual Device...**。

    i. 点击Category列表的**Wear**选项。
    ii. 选择Android Wear Square或者Android Wear Round。
    iii. 点击**Next**按钮。
    iv. 选择一个release name（例如，KitKat Wear）。
    v. 点击**Next**按钮。
    vi. （可选）改变虚拟设备的首选项。
    vii. 点击**Finish**按钮。
3. 启动模拟器:
    i. 选择我们刚才创建的虚拟设备。
    ii. 点击**Play**按钮。
    iii. 等待模拟器初始化直到显示Android Wear的主界面。
4. 匹配手持和模拟器:
    i. 在我们的手持设备上，从Google Play安装Android Wear应用。
    ii. 通过USB将手持设备连接到电脑。
    iii. 切换AVD的通信端口到已连接的手持设备(每次连接上手持设备时都要执行这个步骤)：

    ```
    adb -d forward tcp:5601 tcp:5601
    ```

    iv. 启动手持设备上的Android Wear应用，并连接到模拟器。
    v. 点击Android Wear应用右上角的菜单，选择**Demo Cards**。
    vi. 我们选择的卡片会以Notification的形式呈现在模拟器的主页上。

搭建**Android Wear**真机

建立Android Wear真机，需要下面几个步骤：

1. 在手持设备的Google Play上安装Android Wear应用。
2. 按照应用的命令指示与我们的可穿戴设备进行配对。如果你有做建立notification的操作，这个步骤刚好可以测试这一功能。
3. 保持Android Wear应用在手机上的打开状态。
4. 打开Android Wear设备的adb调试开关。
    i. 选择**Settings > About**。
    ii. 点击**Build number** 7次。
    iii. 右滑返回到Setting菜单。
    iv. 进入屏幕底部的**Developer options**。
    v. 点击**ADB Debugging**来打开adb。
5. 通过USB连接可穿戴设备到电脑上，这样我们能够直接安装应用到可穿戴设备上。此时，在可穿戴设备与Android Wear应用上会显示一个消息，提示是否允许进行调试。
6. 在Android Wear应用上，选择**Always allow from this computer**并且点击**OK**。

Android Studio上的**Android** Tool窗口可以显示可穿戴设备的日志。当你执行 `adb devices` 命令的时候，可穿戴设备应该会出现在该窗口中。

创建工程

在开始开发之前，需要创建一个包含可穿戴应用与手持应用这两个模块的工程。在Android Studio中，点击**File** > **New Project**，然后按照创建工程的指引进行操作。在我们按照安装向导操作的过程中，输入下面的信息：

1. 在**Configure your Project**窗口里，输入应用的名称与一个包名。
2. 在**Form Factors**窗口中:
    - 勾选**Phone and Tablet**并在**Minimum SDK**下拉菜单中选择**API 9: Android 2.3 (Gingerbread)**。
    - 勾选**Wear**并在**Minimum SDK**下拉菜单中选择**API 20: Android 4.4 (KitKat Wear)**。
3. 在第一个**Add an Activity**窗口，为手机应用添加一个空白的activity。
4. 在第二个**Add an Activity**窗口，为可穿戴应用添加一个空白的activity。

当安装向导完成后，Andorid Studio创建了一个包含**mobile**与**wear**两个模块的工程。现在，我们有一个工程可以在手持设备和可穿戴设备应用中创建activity，service，layout等。在手持应用里面，需要承担大部分繁重的任务，例如网络请求，密集计算任务或者是需要大量用户交互的任务。待这些任务完成之后，通常会把任务结果通过notification发送给可穿戴设备上，或者是通过同步机制发送数据给可穿戴设备。

> **Note:** 可穿戴模块包含了一个"Hello World"的activity，它是使用 `WatchViewStub` 类。该类根据设备屏幕是圆的还是方的来填充一个布局。 `WatchViewStub` 类是wearable support library中的一个UI组件。

安装可穿戴应用

在开发过程中，我们可以像安装手持应用一样直接将应用安装到可穿戴设备上。可以使用 `adb install` 命令，也可以使用Android Studio上面的**Play**按钮。

当需要把应用发布给用户的时候，需要把可穿戴应用打包到手持应用中。当用户从Google Play安装手持应用时，连接上的可穿戴设备会自动收到可穿戴应用。

> **Note:** 如果我们给应用签名是debug key，是无法完成自动安装可穿戴应用的（只有release key才可以）。请参考打包可穿戴应用获取更多信息，学习如何正确的打包。

为了安装"Hello World"应用到可穿戴设备，在Android Studiod的**Run/Debug configuration**的下拉菜单中选中**wear**，点击**Play**按钮即可。在可穿戴设备上会显示activity并打印"Hello world!"

include正确的libraries

项目安装向导会自动把合适的模块依赖添加到对应的 `build.gradle` 文件中。然而，这些依赖并不是必须的，请阅读下面描述判断是否需要这些依赖。

Notifications

The Android v4 support library (或者v13)包含一些API，这些API可以将手持设备应用已经存在的notification扩展到可穿戴应用上。

对于只显示在可穿戴设备上的notification(这意味着，他们是由直接执行在可穿戴设备上的app进行处理的)，我们可以在Wear模块仅仅使用标准APIs (API Level 20) 并且把Mobile模块的support library依赖移除。

Wearable Data Layer

可穿戴与手持设备之间进行同步与发送数据需要使用Wearable Data Layer APIs, 这需要用到最新版本的Google Play Services。如果我们不需要这些APIs，可以从这两个模块中把这部分的依赖移除。

Wearable UI support library

这是一个非官方正式的library，它包含了为可穿戴设备设计的UI组件。我们鼓励你在你的应用中使用他们，因为这些组件是最佳实践的例证。但是他们可能随时发生变化。然而，如果library有更新，你的应用并不会发送崩溃，因为那些代码已经编译到你的应用中了。为了获取更新包中新的功能，你只需要更新链接到新的版本并相应的更新你的应用就好了。这个library只是在你需要创建可穿戴应用时才会使用到。

在下一节课，我们将会学习如何创建为可穿戴设备设计的布局，同时学习如何使用各种语音action。

创建自定义的布局

> 编写: kesenhoo - 原文: http://developer.android.com/training/wearables/apps/layouts.html

为可穿戴设备创建布局是和手持设备是一样的，除了我们需要为屏幕的尺寸和glanceability进行设计。但是不要期望通过搬迁手持应用的功能与UI到可穿戴上会有一个好的用户体验。仅仅在有需要的时候，我们才应该创建自定义的布局。请参考可穿戴设备的design guidelines学习如何设计一个优秀的可穿戴应用。

创建自定义Notification

通常来说，我们应该在手持应用上创建好notification，然后让它自动同步到可穿戴设备上。这让我们只需要创建一次notification，然后可以在不同类型的设备(不仅仅是可穿戴设备，也包含车载设备与电视)上进行显示，免去为不同设备进行重新设计。

如果标准的notification风格无法满足我们的需求(例如NotificationCompat.BigTextStyle 或者 NotificationCompat.InboxStyle)，我们可以显示一个使用自定义布局的activity。我们只可以在可穿戴设备上创建并处理自定义的notification，同时系统不会将这些notification同步到手持设备上。

Note: 当在可穿戴设备上创建自定义的notification时，我们可以使用标准notification API（API Level 20），不需要使用Support Library。

为了创建自定义的notification，步骤如下：

1. 创建布局并设置这个布局为需要显示的activity的content view:

```
public void onCreate(Bundle bundle){
 ...
 setContentView(R.layout.notification_activity);
}
```

2. 为了使得activity能够显示在可穿戴设备上，需要在manifest文件中为activity定义必须的属性。我们需要把activity声明为exportable，embeddable以及拥有一个空的task affinity。我们也推荐把activity的主题设置为 `Theme.DeviceDefault.Light` 。例如：

```
<activity android:name="com.example.MyDisplayActivity"
  android:exported="true"
  android:allowEmbedded="true"
  android:taskAffinity=""
  android:theme="@android:style/Theme.DeviceDefault.Light" />
```

3. 为activity创建PendingIntent，例如：：

```
Intent notificationIntent = new Intent(this, NotificationActivity.class);
PendingIntent notificationPendingIntent = PendingIntent.getActivity(this, 0, notif
icationIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
```

4. 创建Notification并执行setDisplayIntent())方法，参数是前面创建的PendingIntent。当用户查看这个notification时，系统使用这个PendingIntent来启动activity。
5. 使用notify())方法触发notification。

> **Note:** 当notification呈现在主页时，系统会根据notification的语义，使用一个标准的模板来呈现它。这个模板可以在所有的表盘上进行显示。当用户往上滑动notification时，将会看到为这个notification准备的自定义的activity。

使用Wearable UI库创建布局

当我们使用Android Studio的工程向导创建一个Wearable应用的时候，会自动包含Wearable UI库。你也可以通过给 `build.gradle` 文件添加下面的依赖声明把库文件添加到项目：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:+'
    compile 'com.google.android.gms:play-services-wearable:+'
}
```

这个库文件帮助我们建立为可穿戴设备设计的UI。更详细的介绍请看为可穿戴设备创建自定义UI。

下面是一些Wearable UI库中主要的类：

- **BoxInsetLayout** - 一个能够感知屏幕的形状并把子控件居中摆放在一个圆形屏幕的FrameLayout。
- **CardFragment** - 一个能够可拉伸，垂直可滑动卡片的fragment。
- **CircledImageView** - 一个圆形的image view。
- **ConfirmationActivity** - 一个在用户完成一个操作之后用来显示确认动画的activity。
- **CrossFadeDrawable** - 一个drawable。该drawable包含两个子drawable和提供方法来调整这两个子drawable的融合方式。
- **DelayedConfirmationView** - 一个view。提供一个圆形倒计时器，这个计时器通常用于在一段短暂的延迟结束后自动确认某个操作。
- **DismissOverlayView** - 一个用来实现长按消失的View。
- **DotsPageIndicator** - 一个为GridViewPager提供的指示标记，用于指定当前页面相对于所有页面的位置。

- **GridViewPager -** 一个可以横向与纵向滑动的局部控制器。你需要提供一个GridPagerAdapter用来生成显示页面的数据。
- **GridPagerAdapter -** 一个提供给GridViewPager显示页面的adapter。
- **FragmentGridPagerAdapter -** 一个将每个页面表示为一个fragment的GridPagerAdapter实现。
- **WatchViewStub -** 一个可以根据屏幕的形状生成特定布局的类。
- **WearableListView -** 一个针对可穿戴设备优化过后的ListView。它会垂直的显示列表内容，并在用户停止滑动时自动显示最靠近的Item。

Wear UI library API reference

这个参考文献解释了如何详细地使用每个UI组件。查看Wear API reference documentation了解上述类的用法。

为用于Eclipse ADT下载Wearable UI库

如果你正在使用Eclipse ADT，那么下载Wearable UI library将Wearable UI库导入到你的工程当中。

> **Note:** 我们推荐使用Android Studio来开发可穿戴应用。

添加语音功能

> 编写: kesenhoo - 原文: http://developer.android.com/training/wearables/apps/voice.html

语音指令是可穿戴体验的一个重要的部分。这使得用户可以释放双手，快速发出指令。穿戴提供了2种类型的语音操作：

- 系统提供的

  这些语音指令都是基于任务的，并且内置在Wear的平台内。我们在activity中过滤我们想要接收的指令。例如包含"Take a note" 或者 "Set an alarm"的指令。

- 应用提供的

  这些语音指令都是基于应用的，我们需要像声明一个Launcher Icon一样声明这些指令。用户通过说"Start "来使用那些语音指令，然后会启动我们指定启动的activity。

声明系统提供的语音指令

Android Wear平台基于用户的操作提供了一些语音指令，例如"Take a note" 或者 "Set an alarm"。用户发出想要做的操作指令，让系统启动最合适的activity。

当用户说出语音指令时，我们的应用能够过滤出用于启动activity的intent。如果我们想要启动一个在后台执行任务的service，需要显示一个activity作为视觉线索，并且在该activity中启动service。当我们想要废弃这个视觉线索时，需要确保执行了finish()。

例如，对于"Take a note"的指令，声明下面这个intent filter来启动一个名为 `MyNoteActivity` 的activity:

```
<activity android:name="MyNoteActivity">
      <intent-filter>
          <action android:name="android.intent.action.SEND" />
          <category android:name="com.google.android.voicesearch.SELF_NOTE" />
      </intent-filter>
  </activity>
```

下面列出了Wear平台支持的语音指令：

Name	Example Phrases	Intent
Call a car/taxi	"OK Google, get me a taxi" "OK Google, call me a car"	Action `com.google.android.gms.actions.RESERVE_TAXI_RESERVATION`

Take a note	"OK Google, take a note"
	
	"OK Google, note to self"
		Action
		`android.intent.action.SEND`
		Category
		`com.google.android.voicesearch.SELF_NOTE`
		Extras
		`android.content.Intent.EXTRA_TEXT` - a string with note body

Set alarm	"OK Google, set an alarm for 8 AM" "OK Google, wake me up at 6 tomorrow"	Action `android.intent.action.SET_ALARM` Extras `android.provider.AlarmClock.EXTRA_HOUR` - an integer with the hour of the alarm. `android.provider.AlarmClock.EXTRA_MINUTES` - an integer with the minute of the alarm (these 2 extras are optional, either none or both are provided)
Set timer	"Ok Google, set a timer for 10 minutes"	Action `android.provider.AlarmClock.ACTION_SET_TIMER` Extras `android.provider.AlarmClock.EXTRA_LENGTH` - an integer in the range of 1 to 86400 (number of seconds in 24 hours) representing the length of the timer

Start/Stop a bike ride	"OK Google, start cycling"
	
	"OK Google, start my bike ride"
	
	"OK Google, stop cycling"
		Action
		`vnd.google.fitness.TRACK`
		Mime Type
		`vnd.google.fitness.activity/biking`
		Extras
		`actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping.

Start/Stop a run	"OK Google, track my run"	Action `vnd.google.fitness.TRACK` MimeType

	"OK Google, start running" "OK Google, stop running"	`vnd.google.fitness.activity/running` Extras `actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

Start/Stop a workout	"OK Google, start a workout"
	
	"OK Google, track my workout"
	
	"OK Google, stop workout"
		Action
		`vnd.google.fitness.TRACK`
		
		MimeType
		`vnd.google.fitness.activity/other`
		
		Extras
		`actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

Show heart rate	"OK Google, what's my heart rate?"
	
	"OK Google, what's my bpm?"
		Action
		`vnd.google.fitness.VIEW`
		
		Mime Type
		`vnd.google.fitness.data_type/com.google.heart_rate.bpm`

Show step count	"OK Google, how many steps have I taken?"
	
	"OK Google, what's my step count?"
		Action
		`vnd.google.fitness.VIEW`
		
		Mime Type
		`vnd.google.fitness.data_type/com.google.step_count.cumulative`

关于注册intent与获取intent extra的信息，请参考Common intents.

声明应用提供的语音指令

如果系统提供的语音指令无法满足我们的需求，我们可以使用"Start MyActivityName"语音指令来直接启动我们的应用。

注册一个"Start"指令的方法和注册手持应用上的Launcher Icon是一样的。除了在launcher里面需要一个应用图标，而我们的应用需要一个语音指令。

为了指定在"Start"指令之后需要说出的文本，我们需要指定想要启动的activity的 `label` 属性。例如，下面的intent filter能够识别"Start MyRunningApp"语音指令并启动 `StartRunActivity` 。

```
<application>
  <activity android:name="StartRunActivity" android:label="MyRunningApp">
      <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
  </activity>
</application>
```

获取自由格式的语音输入

除了使用语音指令来启动activity之外，我们也可以执行系统内置的语言识别activity来获取用户的语音输入。这对于获取用户的输入信息非常有帮助，例如执行搜索或者发送一个消息。

在我们的应用中，使用ACTION_RECOGNIZE_SPEECH action并调用startActivityForResult()。这样可以启动系统语音识别应用，并且我们可以在onActivityResult()中处理返回的结果：

```
private static final int SPEECH_REQUEST_CODE = 0;

// Create an intent that can start the Speech Recognizer activity
private void displaySpeechRecognizer() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
            RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
// Start the activity, the intent will be populated with the speech text
    startActivityForResult(intent, SPEECH_REQUEST_CODE);
}

// This callback is invoked when the Speech Recognizer returns.
// This is where you process the intent and extract the speech text from the intent.
@Override
protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {
    if (requestCode == SPEECH_REQUEST && resultCode == RESULT_OK) {
        List<String> results = data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);
        String spokenText = results.get(0);
        // Do something with spokenText
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

打包可穿戴应用

> 编写: kesenhoo - 原文: http://developer.android.com/training/wearables/apps/packaging.html

当发布应用给用户之前，我们必须把可穿戴应用打包到手持应用内。因为用户不能直接在可穿戴设备上浏览并安装应用。如果打包正确，当用户下载手持应用时，系统会自动下发可穿戴应用到配对好的可穿戴设备上。

> **Note:** 如果开发时签名用的是debug key，这个功能是无法正常工作的。在开发时，需要使用 `adb install` 命令或者Android Studio来安装可穿戴应用。

使用Android Studio打包

在Android Studio中打包可穿戴应用有下面几个步骤：

1. 在手持设备应用的manifest文件中包括所有在可穿戴设备应用manifest文件中声明的权限。例如，如果我们在可穿戴应用中指定了VIBRATE权限，那么我们必须将该权限添加到手持设备应用中。
2. 确保可穿戴应用和手持应用都有相同的包名和版本号。
3. 在手持应用的 `buidl.gradle` 文件中声明一个Gradle依赖用于指向可穿戴应用：

```
dependencies {
compile 'com.google.android.gms:play-services:5.0.+@aar'
compile 'com.android.support:support-v4:20.0.+''
wearApp project(':wearable')
}
```

4. 点击**Build > Generate Signed APK...**，按照屏幕上的指示来制定我们的release key并为我们的app进行签名。Android Studio将签名好的内置了可穿戴应用的手持应用自动导出到工程的根目录。或者，我们可以使用Gradle wrapper在命令行下为在可穿戴应用与手持应用签名。为了能够正常自动推送可穿戴应用，这两个应用都必须签名。将我们的key文件位置和凭证保存到环境变量中，然后如下运行Gradle wrapper：

```
./gradlew assembleRelease \
-Pandroid.injected.signing.store.file=$KEYFILE \
-Pandroid.injected.signing.store.password=$STORE_PASSWORD \
-Pandroid.injected.signing.key.alias=$KEY_ALIAS \
-Pandroid.injected.signing.key.password=$KEY_PASSWORD
```

分别为可穿戴应用与手持应用进行签名

如果我们的构建过程需要将可穿戴应用的签名与手持应用的分开，那么我们可以像下面一样在手持应用的 `build.gradle` 文件中声明Gradle规则。从而嵌入预先签名的可穿戴应用：

```
dependencies {
  ...
  wearApp files('/path/to/wearable_app.apk')
}
```

我们可以以任何我们想要的方式为手持应用进行签名（可以是Android Studio **Build > Generate Signed APK...**的方式，也可以是Gradle `signingConfig` 规则的方式）。

手动打包

如果我们使用的是其它IDE或者其它方法来构建应用，我们仍然可以手动地把可穿戴应用打包到手持应用中。

1. 在手机应用的manifest文件中包括所有在可穿戴设备应用manifest文件中声明的权限。例如，如果我们在可穿戴应用中指定了VIBRATE权限，那么我们必须将该权限添加到手机应用中。
2. 确保可穿戴应用和手持应用的APK都有相同的包名和版本号。
3. 把签好名的可穿戴应用放到手持应用工程的 `res/raw` 目录下。我们假设这个APK名为 `wearable_app.apk` 。
4. 创建 `res/xml/wearable_app_desc.xml` 文件，里面包含可穿戴设备的版本信息与路径。例如:

```
<wearableApp package="wearable.app.package.name">
<versionCode>1</versionCode>
<versionName>1.0</versionName>
<rawPathResId>wearable_app</rawPathResId>
</wearableApp>
```

   `package` , `versionCode` 与 `versionName` 需要和可穿戴应用的AndroidManifest.xml里面的信息一致。 `rawPathResId` 是一个静态变量表示APK的名称。例如，对于 `wearable_app.apk` ，这个静态变量名为 `wearable_app` 。
5. 添加 `meta-data` 标签到我们的手持应用的 `<application>` 标签下，指明引用 `wearable_app_desc.xml` 文件

```
<meta-data android:name="com.google.android.wearable.beta.app"
              android:resource="@xml/wearable_app_desc"/>
```

6. 构建并签名手持应用。

关闭资源压缩

许多构建工具会自动压缩放在 `res/raw` 目录下的文件。因为可穿戴APK已经被压缩过了，所以这些工具再次压缩可穿戴APK会导致可穿戴应用安装程序无法读取可穿戴应用。

这样的话，安装失败。在手持应用上， `PackageUpdateService` 会输出如下的错误日志："this file cannot be opened as a file descriptor; it is probably compressed."

Android Studio 默认不会压缩APK，但是如果我们使用其它构建方式，需要确保不要重复压缩可穿戴应用。

通过蓝牙进行调试

> 编写: [kesenhoo](https://github.com/kesenhoo) - 原文: http://developer.android.com/training/wearables/apps/bt-debugging.html

我们可以通过蓝牙来调试我们的可穿戴应用。即通过蓝牙把调试数据输出到已经连接了开发电脑的手持设备上。

搭建好设备用来调试

1. 开启手持设备的USB调试：
   - 打开设置应用并滑动到底部。
   - 如果在设置里面没有开发者选项，点击关于手机（或者关于平板），滑动到底部，点击build number 7次。
   - 返回并点击开发者选项。
   - 开启**USB**调试。
2. 开启可穿戴设备的蓝牙调试：
   - 点击主界面2次，来到Wear菜单界面。
   - 滑动到底部，点击设置。
   - 滑动到底部，如果没有开发者选项，点击关于，然后点击Build Number 7次。
   - 点击开发者选项。
   - 开启蓝牙调试。

建立调试会话

1. 在手持设备上，打开 `Android Wear` 配套应用。
2. 点击右上角的菜单，选择设置。
3. 开启蓝牙调试。我们将会在选项下面看到一个小的状态信息：

```
Host: disconnected
Target: connected
```

4. 通过USB连接手持设备到电脑上，并执行下面的命令：

```
adb forward tcp:4444 localabstract:/adb-hub
adb connect localhost:4444
```

> **Note:** 我们可以使用任何可用的端口。

在 `Android Wear` 配套应用上，我们将会看到状态变为：

```
Host: connected
Target: connected
```

调试应用

当运行 `abd devices` 的命令时，我们的可穿戴设备应该表示为localhost:4444。执行任何的 `adb` 命令，需要使用下面的格式：

```
adb -s localhost:4444 <command>
```

如果没有任何其他的设备通过TCP/IP连接到手持设备（即模拟器），我们可以使用下面的简短命令：

```
adb -e <command>
```

例如：

```
adb -e logcat
adb -e shell
adb -e bugreport
```

为可穿戴设备创建自定义UI

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/index.html

可穿戴apps的用户界面明显的不同于手持设备。可穿戴设备应用应该参考Android Wear设计规范和实现推荐的UI模式， 这些保证在为可穿戴设备优化过的应用中保持统一的用户体验。

这个课程将教我们如何为可穿戴应用创建在所有Android可穿戴设备上看上去都不错的自定义UI和自定义的notifications。为了达到上述目的，需要实现这些UI模式：

- Card
- 倒计时和确认
- 长按退出
- 2D Picker
- 多选List

可穿戴UI库是Android SDK的Google Repository中的一部分，其中提供的类可以帮助我们实现这些模式和创建工作在圆形和方形Android可穿戴设备的layout。

> **Note:** 我们推荐使用Android Studio做Android Wear开发,它提供工程初始配置,库包含和方便的打包,这些在ADT中是没有的。这系列教程假定你正在使用Android Studio。

Lessons

定义Layouts

学习如何创建在圆形和方形Android Wear设备上看起来不错的layout。

创建Card

学习如何创建自定义layout的Card

创建List

学习如何创建为可穿戴设备优化的List

创建2D Picker

学习如何实现2D Picker UI模式以导航各页数据

显示确认界面

学习如何在用户完成操作时显示确认动画

退出全屏的Activity

学习如何实现长按退出UI模式以退出全屏activities

定义Layouts

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/layouts.html

可穿戴设备使用与手持Android设备同样的布局技术，但需要有具体的约束来设计。不要以一个手持app的角度开发功能和UI并期待得到一个好的体验。关于如何设计优秀的可穿戴应用的更多信息，请阅读Android Wear Design Guidelines。

当为Android Wear应用创建layout时，我们需要同时考虑方形和圆形屏幕的设备。在圆形Android Wear设备上所有放置在靠近屏幕边角的内容可能会被剪裁掉，所以为方形屏幕设计的layouts在圆形设备上不能很好地显示出来。对这类问题的示范请查看这个视频Full Screen Apps for Android Wear。

举个例子，figure 1展示了下面的layout在圆形和方形屏幕上的效果：



Figure 1. 为方形屏幕设计的layouts在圆形设备上不能很好显示的示范

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_square" />
</LinearLayout>
```

上述范例的文本没有正确地显示在圆形屏幕上。

Wearable UI库为这个问题提供了两种不同的解决方案：

- 为圆形和方形屏幕定义不同的layouts。我们的app会在运行时检查设备屏幕形状并inflate正确的layout。
- 用一个包含在库里面的特殊layout同时适配方形和圆形设备。这个layout会在不同形状的设备屏幕窗口中插入不同的间隔。

当我们希望应用在不同形状的屏幕上看起来不同时，一般会使用第一种方案。当我们希望用一个相似的layout在两种屏幕上且在圆形屏幕上没有视图被边缘剪裁时，可以使用第二种方案。

添加Wearable UI库

当我们使用Android Studio的工程向导时，Android Studio会自动地在 `wear` 模块中包含Wearable UI库。为了在工程中编译到这个库，确保 *Extras* > *Google Repository* 包已经被安装在Android SDK manager里，下面的依赖被包含在 `wear` 模块的 `build.gradle` 文件中：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:+'
    compile 'com.google.android.gms:play-services-wearable:+'
}
```

要实现以下的布局方法需要用到 `'com.google.android.support:wearable'` 依赖。

浏览API reference documentation查看Wearable UI库的类。

为方形和圆形屏幕指定不同的Layouts

包含在Wearable UI库中的 `WatchViewStub` 类允许我们为方形和圆形屏幕指定不同的layout。这个类会在运行时检查屏幕形状并inflate相应的layout。

为了在我们的应用中使用这个类以应对不同的屏幕形状，我们需要：

- 添加 `WatchViewStub` 作为activity的layout的主元素。
- 使用 `rectLayout` 属性为方形屏幕指定一个layout文件。
- 使用 `roundLayout` 属性为圆形屏幕指定一个layout文件。

类似下面定义activity的layout：

```
<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/watch_view_stub"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_wear"
    app:roundLayout="@layout/round_activity_wear">
</android.support.wearable.view.WatchViewStub>
```

在activity中inflate这个layout：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear);
}
```

然后为方形和圆形屏幕创建不同的layout文件，在这个例子中，我们需要创建 `res/layout/rect_activity_wear.xml` 和 `res/layout/round_activity_wear.xml` 两个文件。像创建手持应用的layouts一样定义这些layouts，但需要考虑可穿戴设备的限制。系统会在运行时根据屏幕形状来inflate适合的layout。

取得layout views

我们为方形或圆形屏幕定义的layouts在 `WatchViewStub` 检测到屏幕形状之前不会被inflate，所以你的app不能立即取得它们的view。为了取得这些view，需要在我们的activity中设置一个listener，当屏幕适配的layout被inflate时会通知这个listener：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear);

    WatchViewStub stub = (WatchViewStub) findViewById(R.id.watch_view_stub);
    stub.setOnLayoutInflatedListener(new WatchViewStub.OnLayoutInflatedListener() {
        @Override public void onLayoutInflated(WatchViewStub stub) {
            // Now you can access your views
            TextView tv = (TextView) stub.findViewById(R.id.text);
            ...
        }
    });
}
```

使用感知形状的Layout

包含在Wearable UI库中的 `BoxInsetLayout` 类继承自 FrameLayout，该类允许我们定义一个同时适配方形和圆形屏幕的layout。这个类适用于需要根据屏幕形状插入间隔的情况，并让我们容易地将view对其到屏幕的边缘或中心。



Figure 2. 在圆形屏幕上的窗口间隔

figure 2中灰色的部分显示了在应用了窗口间隔之后，`BoxInsetLayout` 自动将它的子view放置在圆形屏幕的区域。为了显示在这个区域内，子view需要用下面这些值指定 `layout_box` 属性：

- 一个 `top` 、 `bottom` 、 `left` 和 `right` 的组合。比如， `"left|top"` 将子view的左和上边缘定位在figure 2的灰色区域里面。
- `all` 将所有子view的内容定位在figure 2的灰色区域里面。

在方形屏幕上，窗口间隔为0， `layout_box` 属性会被忽略。

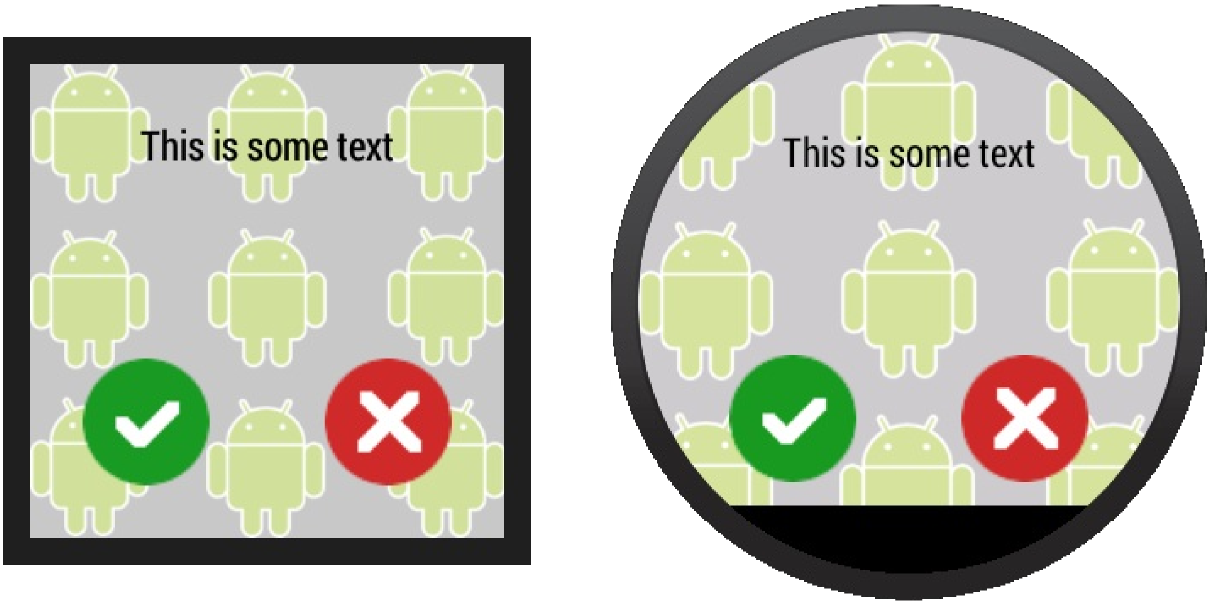


Figure 3. 同一个layout工作在方形和圆形屏幕上

在figure 3中展示的layout使用了 `BoxInsetLayout` ，该layout在圆形和方形屏幕上都可以使用：

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="@drawable/robot_background"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:padding="15dp">

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp"
        app:layout_box="all">

        <TextView
            android:gravity="center"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:text="@string/sometext"
            android:textColor="@color/black" />

        <ImageButton
            android:background="@null"
            android:layout_gravity="bottom|left"
            android:layout_height="50dp"
            android:layout_width="50dp"
            android:src="@drawable/ok" />

        <ImageButton
            android:background="@null"
            android:layout_gravity="bottom|right"
            android:layout_height="50dp"
            android:layout_width="50dp"
            android:src="@drawable/cancel" />
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

注意layout中的这些部分：

- `android:padding="15dp"`

这行指定了 `BoxInsetLayout` 元素的padding。因为在圆形设备上窗口间隔大于15dp，所以这个padding只应用在方形屏幕上。

- `android:padding="5dp"`

这行指定内部 `FrameLayout` 元素的padding。这个padding同时应用在方形和圆形屏幕上。在方形屏幕上，按钮和窗口间隔总的padding是20dp(15+5)，在圆形屏幕上是5dp。

- `app:layout_box="all"`

这行声明 `FrameLayout` 和它的子views都被放在圆形屏幕上窗口间隔定义的区域里。这行在方形屏幕上没有任何效果。

创建Card

> 编写: [roya](https://github.com/roya) 原文:https://developer.android.com/training/wearables/ui/cards.html

Card在不同的应用上以一致的外观为用户显示信息。这个章节介绍如何在Android Wear应用中创建Card。

Wearable UI库提供了为穿戴设备特别设计的Card实现。这个库包含了 `CardFrame` 类，它将view包在一个Card风格的框架中，该框架有白色的背景、圆角和光投射阴影。 `CardFrame` 只能包含一个直接子类，通常是一个layout管理器，我们可以向它添加其他views以定制Card内容。

你有两种方法向应用添加Card：

- 使用或继承 `CardFragment` 类。
- 在layout的 `CardScrollView` 中添加一个Card。

> **Note:** 这个课程展示了如何在Android Wear activities中添加Card。Android可穿戴设备上的notifications同样以Card的形式显示。更多信息请查看为Notification赋加可穿戴特性。

创建Card Fragment

`CardFragment` 类提供一个默认的Card layout，该layout含有一个标题、描述文字和一个图标。如果figure 1的默认Card layout符合你的要求，那么使用这个方法向你的app添加Card。

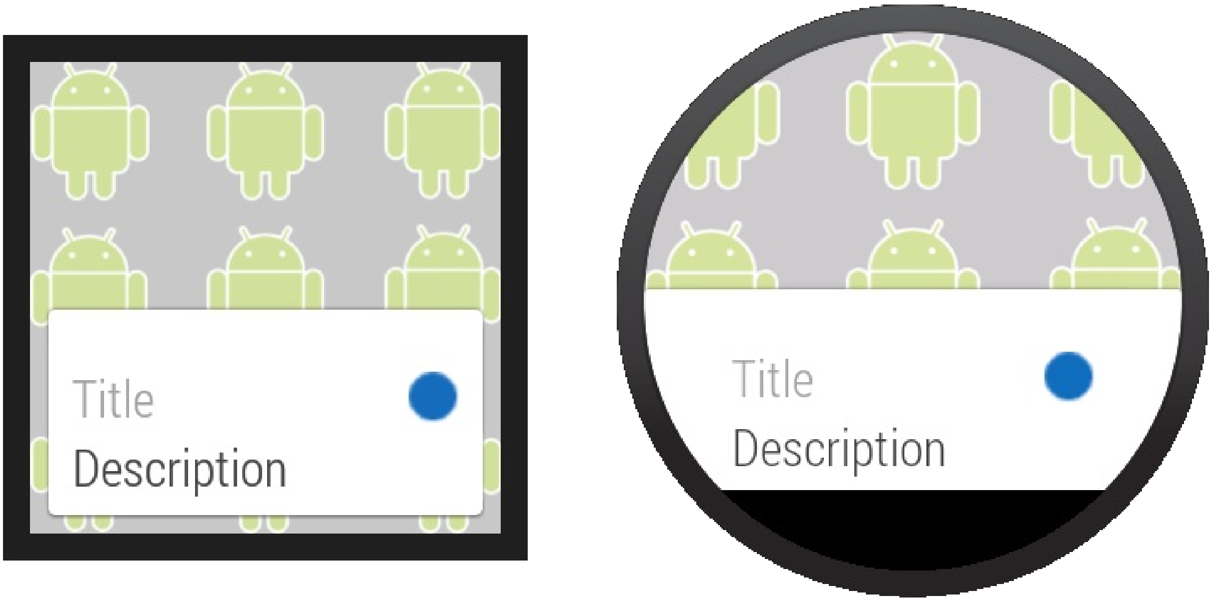


Figure 1. 默认的 `CardFragment` layout.

为了添加一个 `CardFragment` 到应用中，我们需要：

- 在layout中，为包含Card的节点分配一个ID
- 在activity中，创建一个 `CardFragment` 实例
- 使用fragment管理器将 `CardFragment` 实例添加到它的容器

下面的示例代码显示了Figure 1中的屏幕显示代码：

```
<android.support.wearable.view.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:background="@drawable/robot_background"
android:layout_height="match_parent"
android:layout_width="match_parent">

    <FrameLayout
        android:id="@+id/frame_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_box="bottom">

    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

下面的代码添加 `CardFragment` 实例到Figure 1的activity中：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear_activity2);

    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
    CardFragment cardFragment = CardFragment.create(getString(R.string.cftitle),
                                                    getString(R.string.cfdesc),
                                                    R.drawable.p);
    fragmentTransaction.add(R.id.frame_layout, cardFragment);
    fragmentTransaction.commit();
}
```

为了使用 `CardFragment` 创建一个带有自定义layout的Card，需要继承这个类和重写它的 `onCreateContentView` 方法。

添加CardFrame到Layout

我们也可以直接添加一个Card到layout中，如figure 2所示。当希望为layout文件中的Card自定义一个layout时，使用这个方法。

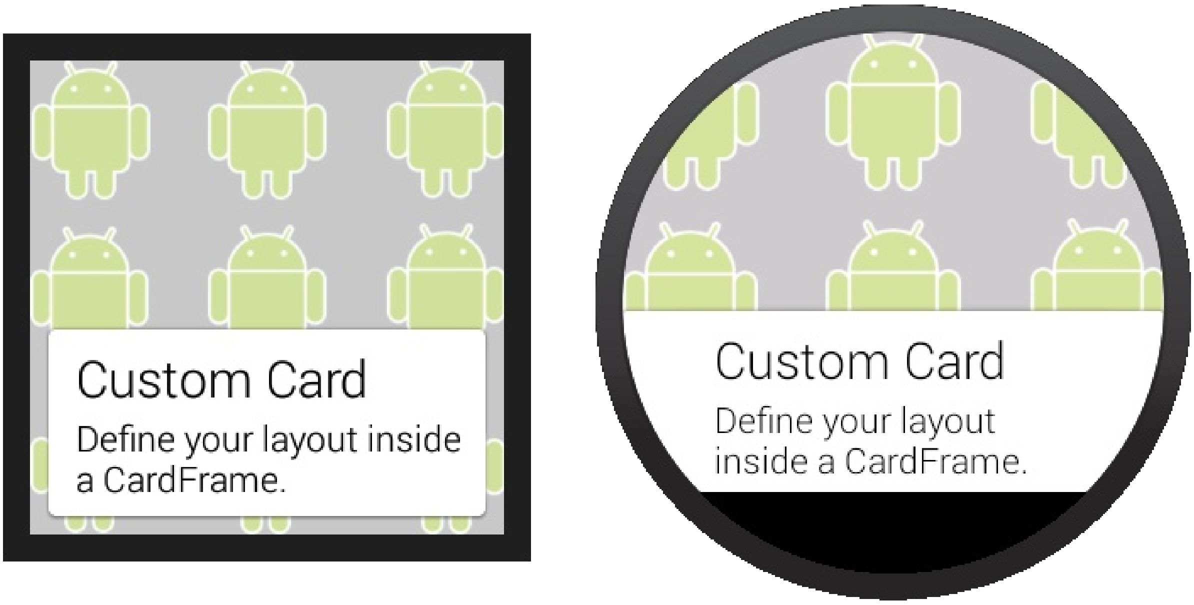


Figure 2. 添加一个 `CardFrame` 到layout.

下面的layout代码例子示范了一个含有两个节点的垂直linear layout。你可以创建更加复杂的layouts以适合你应用的需要。

```
<android.support.wearable.view.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:background="@drawable/robot_background"
android:layout_height="match_parent"
android:layout_width="match_parent">

    <android.support.wearable.view.CardScrollView
        android:id="@+id/card_scroll_view"
        android:layout_height="match_parent"
        android:layout_width="match_parent"
        app:layout_box="bottom">

        <android.support.wearable.view.CardFrame
            android:layout_height="wrap_content"
            android:layout_width="fill_parent">

            <LinearLayout
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:orientation="vertical"
                android:paddingLeft="5dp">
                <TextView
                    android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content"
                    android:layout_width="match_parent"
                    android:text="@string/custom_card"
                    android:textColor="@color/black"
                    android:textSize="20sp"/>
                <TextView
                    android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content"
                    android:layout_width="match_parent"
                    android:text="@string/description"
                    android:textColor="@color/black"
                    android:textSize="14sp"/>
            </LinearLayout>
        </android.support.wearable.view.CardFrame>
    </android.support.wearable.view.CardScrollView>
</android.support.wearable.view.BoxInsetLayout>
```

当 `CardScrollView` 的内容小于容器时，这个例子上的 `CardScrollView` 节点让我们可以配置Card的gravity，。这个例子是Card对齐屏幕底部：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear_activity2);

    CardScrollView cardScrollView =
        (CardScrollView) findViewById(R.id.card_scroll_view);
    cardScrollView.setCardGravity(Gravity.BOTTOM);
}
```

`CardScrollView` 检测屏幕形状后以不同的显示方式在圆形或方形设备上显示Card（在圆形屏幕上使用更宽的侧边缘。不管怎样，在 `BoxInsetLayout` 中放置 `CardScrollView` 节点然后使用 `layout_box="bottom"` 属性，这对圆形屏幕上的Card对齐底部并且没有内容被剪裁是很有用的。

创建List

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/lists.html

List让用户在可穿戴设备上很容易地从一组选项中选择一个项目。这个课程介绍了如何在Android Wear应用中创建List。

Wearable UI库包含了 `WearableListView` 类，该类是对可穿戴设备进行优化的List实现。

> **Note:** Android SDK 中的 `Notifications` 例子示范了如何在应用中使用 `WearableListView` 。这个例子的位于 `android-sdk/samples/android-20/wearable/Notifications` 目录。

为了在Android Wear应用中创建List，我们需要:

1. 添加 `WearableListView` 元素到activity的layout定义中。
2. 为List选项创建一个自定义的layout实现。
3. 使用这个实现为List选项创建一个layout定义文件。
4. 创建一个adapter以填充List。
5. 指定这个adapter到 `WearableListView` 元素。

下面的章节有这些步骤的详细描述。

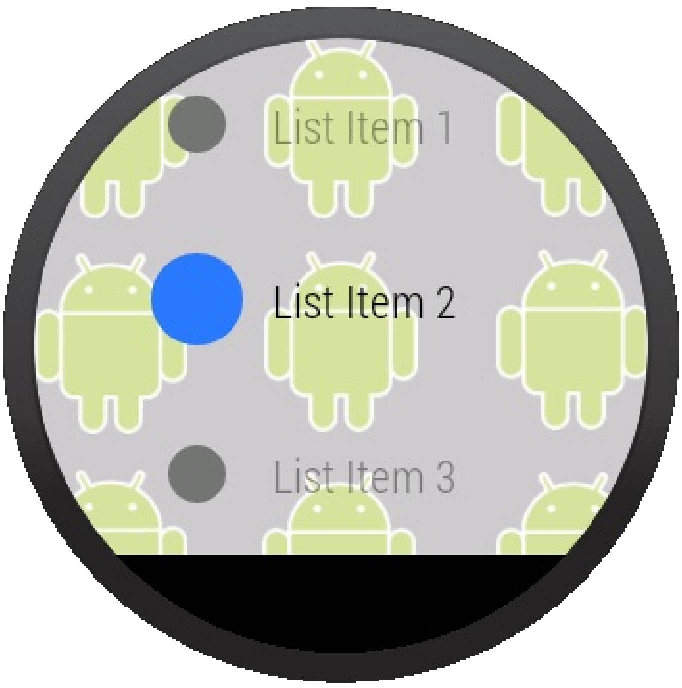


Figure 3: 在Android Wear上的List View.

添加List View

下面的layout使用 `BoxInsetLayout` 添加了一个List view到activity中，所以这个List可以正确地显示在圆形和方形两种设备上：

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="@drawable/robot_background"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <FrameLayout
        android:id="@+id/frame_layout"
        android:layout_height="match_parent"
        android:layout_width="match_parent"
        app:layout_box="left|bottom|right">

        <android.support.wearable.view.WearableListView
            android:id="@+id/wearable_list"
            android:layout_height="match_parent"
            android:layout_width="match_parent">
        </android.support.wearable.view.WearableListView>
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

为List选项创建一个Layou实现

在许多例子中，每个List选项都由一个图标和一个描述组成。Android SDK中的*Notifications*例子实现了一个自定义layout：继承LinearLayout以合并两元素到每个List选项。这个layout也实现了 `WearableListView.OnCenterProximityListener` 接口里的方法，以实现在用户在List中滚动时，因 `WearableListView` 的事件而改变选项图标颜色和渐隐文字：

```
public class WearableListItemLayout extends LinearLayout
             implements WearableListView.OnCenterProximityListener {

    private ImageView mCircle;
    private TextView mName;

    private final float mFadedTextAlpha;
    private final int mFadedCircleColor;
    private final int mChosenCircleColor;

    public WearableListItemLayout(Context context) {
        this(context, null);
    }

    public WearableListItemLayout(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public WearableListItemLayout(Context context, AttributeSet attrs,
                                  int defStyle) {
        super(context, attrs, defStyle);

        mFadedTextAlpha = getResources()
                         .getInteger(R.integer.action_text_faded_alpha) / 100f;
        mFadedCircleColor = getResources().getColor(R.color.grey);
        mChosenCircleColor = getResources().getColor(R.color.blue);
    }

    // Get references to the icon and text in the item layout definition
    @Override
    protected void onFinishInflate() {
        super.onFinishInflate();
        // These are defined in the layout file for list items
        // (see next section)
        mCircle = (ImageView) findViewById(R.id.circle);
        mName = (TextView) findViewById(R.id.name);
    }

    @Override
    public void onCenterPosition(boolean animate) {
        mName.setAlpha(1f);
        ((GradientDrawable) mCircle.getDrawable()).setColor(mChosenCircleColor);
    }

    @Override
    public void onNonCenterPosition(boolean animate) {
        ((GradientDrawable) mCircle.getDrawable()).setColor(mFadedCircleColor);
        mName.setAlpha(mFadedTextAlpha);
    }
}
```

我们也可以创建animator对象以放大List中间选项的图标。我们可以使用 `WearableListView.OnCenterProximityListener` 接口的 `onCenterPosition()` 和 `onNonCenterPosition()` 回调方法来管理animator对象。更多关于animator对象的信息请查看Animating with ObjectAnimator

为Items创建Layout解释

在为List选项实现自定义layout之后，我们需要提供一个layout解释文件以具体说明list item中的组件参数。下面的layout使用先前的自定义layout实现，并且定义图标和文本view，这两个view的ID对应layout实现类的ID：

`res/layout/list_item.xml`

```
<com.example.android.support.wearable.notifications.WearableListItemLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:gravity="center_vertical"
    android:layout_width="match_parent"
    android:layout_height="80dp">
    <ImageView
        android:id="@+id/circle"
        android:layout_height="20dp"
        android:layout_margin="16dp"
        android:layout_width="20dp"
        android:src="@drawable/wl_circle"/>
    <TextView
        android:id="@+id/name"
        android:gravity="center_vertical|left"
        android:layout_width="wrap_content"
        android:layout_marginRight="16dp"
        android:layout_height="match_parent"
        android:fontFamily="sans-serif-condensed-light"
        android:lineSpacingExtra="-4sp"
        android:textColor="@color/text_color"
        android:textSize="16sp"/>
</com.example.android.support.wearable.notifications.WearableListItemLayout>
```

创建Adapter以填充List

Adapter用内容填充 `WearableListView` 。下面的adapter基于strings数组元素填充了List：

```
private static final class Adapter extends WearableListView.Adapter {
    private String[] mDataset;
    private final Context mContext;
    private final LayoutInflater mInflater;
```

```
    // Provide a suitable constructor (depends on the kind of dataset)
    public Adapter(Context context, String[] dataset) {
        mContext = context;
        mInflater = LayoutInflater.from(context);
        mDataset = dataset;
    }

    // Provide a reference to the type of views you're using
    public static class ItemViewHolder extends WearableListView.ViewHolder {
        private TextView textView;
        public ItemViewHolder(View itemView) {
            super(itemView);
            // find the text view within the custom item's layout
            textView = (TextView) itemView.findViewById(R.id.name);
        }
    }

    // Create new views for list items
    // (invoked by the WearableListView's layout manager)
    @Override
    public WearableListView.ViewHolder onCreateViewHolder(ViewGroup parent,
                                                          int viewType) {
        // Inflate our custom layout for list items
        return new ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
    }

    // Replace the contents of a list item
    // Instead of creating new views, the list tries to recycle existing ones
    // (invoked by the WearableListView's layout manager)
    @Override
    public void onBindViewHolder(WearableListView.ViewHolder holder,
                                 int position) {
        // retrieve the text view
        ItemViewHolder itemHolder = (ItemViewHolder) holder;
        TextView view = itemHolder.textView;
        // replace text contents
        view.setText(mDataset[position]);
        // replace list item's metadata
        holder.itemView.setTag(position);
    }

    // Return the size of your dataset
    // (invoked by the WearableListView's layout manager)
    @Override
    public int getItemCount() {
        return mDataset.length;
    }
}
```

连接Adapter和设置Click Listener

在我们的activity中，从layout中取得 `WearableListView` 元素的引用，分配一个adapter实例以填充List，然后设置一个click listener以完成当用户选择了一个特定的List选项的动作。

```
public class WearActivity extends Activity
                          implements WearableListView.ClickListener {

    // Sample dataset for the list
    String[] elements = { "List Item 1", "List Item 2", ... };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_list_activity);

        // Get the list component from the layout of the activity
        WearableListView listView =
            (WearableListView) findViewById(R.id.wearable_list);

        // Assign an adapter to the list
        listView.setAdapter(new Adapter(this, elements));

        // Set a click listener
        listView.setClickListener(this);
    }

    // WearableListView click listener
    @Override
    public void onClick(WearableListView.ViewHolder v) {
        Integer tag = (Integer) v.itemView.getTag();
        // use this data to complete some action ...
    }

    @Override
    public void onTopEmptyRegionClick() {
    }
}
```

创建2D Picker

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/2d-picker.html

Android Wear中的2D Picker模式允许用户像换页一样从一组选项中导航和选择。Wearable UI库让我们可以容易地用一个page grid来实现这个模式。其中，page grid是一个layout管理器，它允许用户垂直和水平滚动页面。

要实现这个模式，我们需要添加一个 `GridViewPager` 元素到activity的layout中，然后实现一个继承 `FragmentGridPagerAdapter` 类的adapter以提供一组页面。

> **Note:** Android SDK中的*GridViewPager*例子示范了如何在应用中使用 `GridViewPager` layout。这个例子的位于 `android-sdk/samples/android-20/wearable/GridViewPager` 目录中。

添加Page Grid

像下面一样添加一个 `GridViewPager` 元素到layout描述文件：

```
<android.support.wearable.view.GridViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

我们可以使用任何定义Layouts技术以保证2D picker可以工作在圆形和方形两种设备上。

实现Page Adapter

Page Adapter提供一组页面以填充 `GridViewPager` 部件。要实现这个adapter，需要继承Wearable UI库中的 `FragmentGridPageAdapter` 类。

举个例子，Android SDK内的*GridViewPager*例子中包含了下面的adapter实现，该实现提供一组静态的具有自定义背景图片的card：

```
public class SampleGridPagerAdapter extends FragmentGridPagerAdapter {

    private final Context mContext;

    public SampleGridPagerAdapter(Context ctx, FragmentManager fm) {
        super(fm);
        mContext = ctx;
    }

    static final int[] BG_IMAGES = new int[] {
            R.drawable.debug_background_1, ...
            R.drawable.debug_background_5
    };

    // A simple container for static data in each page
    private static class Page {
        // static resources
        int titleRes;
        int textRes;
        int iconRes;
        ...
    }

    // Create a static set of pages in a 2D array
    private final Page[][] PAGES = { ... };

    // Override methods in FragmentGridPagerAdapter
    ...
}
```

picker调用 `getFragment` 和 `getBackground` 来取得内容以显示到grid的每个位置中。

```
// Obtain the UI fragment at the specified position
@Override
public Fragment getFragment(int row, int col) {
    Page page = PAGES[row][col];
    String title =
        page.titleRes != 0 ? mContext.getString(page.titleRes) : null;
    String text =
        page.textRes != 0 ? mContext.getString(page.textRes) : null;
    CardFragment fragment = CardFragment.create(title, text, page.iconRes);

    // Advanced settings (card gravity, card expansion/scrolling)
    fragment.setCardGravity(page.cardGravity);
    fragment.setExpansionEnabled(page.expansionEnabled);
    fragment.setExpansionDirection(page.expansionDirection);
    fragment.setExpansionFactor(page.expansionFactor);
    return fragment;
}

// Obtain the background image for the page at the specified position
@Override
public ImageReference getBackground(int row, int column) {
    return ImageReference.forDrawable(BG_IMAGES[row % BG_IMAGES.length]);
}
```

`getRowCount` 方法告诉picker有多少行内容是可获得的，`getColumnCount` 方法告诉picker每行中有多少列内容是可获得的。

```
// Obtain the number of pages (vertical)
@Override
public int getRowCount() {
    return PAGES.length;
}

// Obtain the number of pages (horizontal)
@Override
public int getColumnCount(int rowNum) {
    return PAGES[rowNum].length;
}
```

adapter是实现细节取决于我们指定的某组页面。由adapter提供的每个页面是 `Fragement` 类型。在这个例子中，每个页面是一个使用默认card layouts的 `CardFragment` 实例。然而，我们可以在同一个2D picker混合不同类型的页面，比如cards，action icons，和自定义layouts，由具体情况决定。

不是所有行都需要有同样数量的页面。注意这个例子中的每行有不同的列数。我们也可以用一个 `GridViewPager` 组件实现只有一行或一列的1D picker。

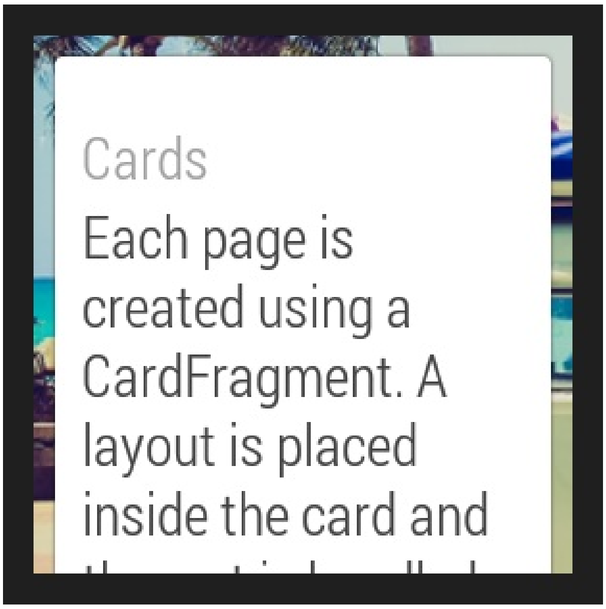


Figure 1: GridViewPager例子

对于那些超出设备屏幕大小的card，`GridViewPager` 为它们提供了滚动支持。这个例子配置了每张card可以按照需要进行展开，所以用户可以滚动卡片的内容。当用户滚动到card的尽头，向同一方向滑动将显示grid中的下一页（如果下一页存在的话）。

我们可以使用 `getBackground()` 方法自定义每页的背景。当用户在页面间滑动时，`GridViewPager` 自动在不同的背景之间使用视差滚动和淡出效果。

分配adapter实例给page grid

在activity中，分配一个adapter实现实例给 `GridViewPager` 组件：

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        final GridViewPager pager = (GridViewPager) findViewById(R.id.pager);
        pager.setAdapter(new SampleGridPagerAdapter(this, getFragmentManager()));
    }
}
```

显示确认界面

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/confirm.html

Android Wear应用中的确认界面(Confirmations)通常是全屏或者相比于手持应用占更大的部分。这样确保用户可以一眼看到确认界面(confirmations)且有一个足够大的触摸区域用于取消一个操作。

Wearable UI库帮助我们在Android Wear应用中显示确认动画和定时器：

确认定时器

- 自动确认定时器为用户显示一个定时器动画，让用户可以取消他们最近的操作。

确认界面动画

- 确认界面动画给用户在完成一个操作时的视觉反馈。

下面的章节将演示了如何实现这些模式。

使用自动确认定时器

自动确认定时器让用户取消刚做的操作。当用户做一个操作，我们的应用会显示一个带有定时动画的取消按钮，并且启动该定时器。用户可以在定时结束前选择取消操作。如果用户选择取消操作或定时结束，我们的应用会得到一个通知。

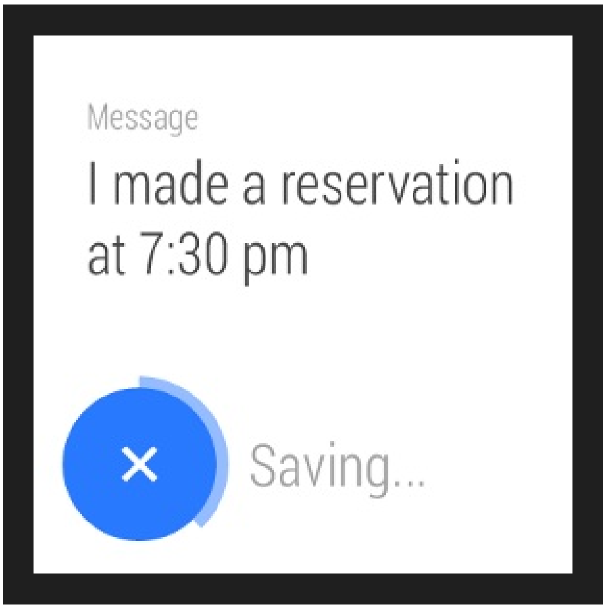


Figure 1: 一个确认定时器.

为了在用户完成操作时显示一个确认定时器：

1. 添加 `DelayedConfirmationView` 元素到layout中。
2. 在activity中实现 `DelayedConfirmationListener` 接口。
3. 当用户完成一个操作时，设置定时器的定时时间然后启动它。

像下面这样添加 `DelayedConfirmationView` 元素到layout中：

```
<android.support.wearable.view.DelayedConfirmationView
    android:id="@+id/delayed_confirm"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:src="@drawable/cancel_circle"
    app:circle_border_color="@color/lightblue"
    app:circle_border_width="4dp"
    app:circle_radius="16dp">
</android.support.wearable.view.DelayedConfirmationView>
```

在layout定义中，我们可以用 `android:src` 制定一个drawable资源，用于显示在圆形里。然后直接设置圆的参数。

为了获得定时结束或用户点击按钮时的通知，需要在activity中实现相应的listener方法：

```
public class WearActivity extends Activity implements
                           DelayedConfirmationView.DelayedConfirmationListener {

    private DelayedConfirmationView mDelayedView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_wear_activity);

        mDelayedView =
                (DelayedConfirmationView) findViewById(R.id.delayed_confirm);
        mDelayedView.setListener(this);
    }

    @Override
    public void onTimerFinished(View view) {
        // User didn't cancel, perform the action
    }

    @Override
    public void onTimerSelected(View view) {
        // User canceled, abort the action
    }
}
```

为了启动定时器，添加下面的代码到activity处理用户选择某个操作的位置中：

```
// Two seconds to cancel the action
mDelayedView.setTotalTimeMs(2000);
// Start the timer
mDelayedView.start();
```

显示确认动画

为了当用户在我们的应用中完成一个操作时显示确认动画，我们需要创建一个从应用中的某个activity启动 `ConfirmationActivity` 的intent。我们可以用 `EXTRA_ANIMATION_TYPE` intent extra来指定下面其中一种动画：

- `SUCCESS_ANIMATION`
- `FAILURE_ANIMATION`
- `OPEN_ON_PHONE_ANIMATION`

我们还可以在确认图标下面添加一条消息。



Figure 2: 一个确认动画

要在应用中使用 `ConfirmationActivity` ，首先在manifest文件声明这个activity：

```
<manifest>
  <application>
    ...
    <activity
        android:name="android.support.wearable.activity.ConfirmationActivity">
    </activity>
  </application>
</manifest>
```

然后确定用户操作的结果，并使用intent启动activity:

```
Intent intent = new Intent(this, ConfirmationActivity.class);
intent.putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE,
                ConfirmationActivity.SUCCESS_ANIMATION);
intent.putExtra(ConfirmationActivity.EXTRA_MESSAGE,
                getString(R.string.msg_sent));
startActivity(intent);
```

当确认动画显示结束后，`ConfirmationActivity` 会销毁（Finish），我们的的activity会恢复（Resume）。

退出全屏的Activity

> 编写: roya 原文:https://developer.android.com/training/wearables/ui/exit.html

默认情况下，用户通过从左到右滑动退出Android Wear activities。如果应用含有水平滚动的内容，用户首先滑动到内容边缘，然后再次从左到右滑动即退出app。

对于更加沉浸式的体验，比如在应用中可以任意方向地滚动地图，这时我们可以在应用中禁用滑动退出手势。然而，如果我们禁用了这个功能，那么我们必须使用Wearable UI库中的 `DismissOverlayView` 类实现长按退出UI模式让用户退出应用。当然，我们需要在用户第一次运行我们应用的时候提醒用户可以通过长按退出应用。

更多关于退出Android Wear activities的设计指南，请查看Breaking out of the card。

禁用滑动退出手势

如果我们应用的用户交互模型与滑动退出手势相冲突，那么我们可以在应用中禁用它。为了禁用滑动退出手势，需要继承默认的theme，然后设置 `android:windowSwipeToDismiss` 属性为 `false` ：

```
<style name="AppTheme" parent="Theme.DeviceDefault">
    <item name="android:windowSwipeToDismiss">false</item>
</style>
```

如果我们禁用了这个手势，那么我们需要实现长按退出UI模型来让用户退出我们的应用，下面的章节会介绍相关内容。

实现长按退出模式

要在activity中使用 `DissmissOverlayView` 类，添加下面这个节点到layout文件，让它全屏且覆盖在所有其他view上。例如：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <!-- other views go here -->

    <android.support.wearable.view.DismissOverlayView
        android:id="@+id/dismiss_overlay"
        android:layout_height="match_parent"
        android:layout_width="match_parent"/>
<FrameLayout>
```

在我们的activity中，取得 `DismissOverlayView` 元素并设置一些提示文字。这些文字会在用户第一次运行我们的应用时提醒用户可以使用长按手势退出应用。接着用 `GestureDetector` 检测长按动作：

```
public class WearActivity extends Activity {

    private DismissOverlayView mDismissOverlay;
    private GestureDetector mDetector;

    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(R.layout.wear_activity);

        // Obtain the DismissOverlayView element
        mDismissOverlay = (DismissOverlayView) findViewById(R.id.dismiss_overlay);
        mDismissOverlay.setIntroText(R.string.long_press_intro);
        mDismissOverlay.showIntroIfNecessary();

        // Configure a gesture detector
        mDetector = new GestureDetector(this, new SimpleOnGestureListener() {
            public void onLongPress(MotionEvent ev) {
                mDismissOverlay.show();
            }
        });
    }

    // Capture long presses
    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        return mDetector.onTouchEvent(ev) || super.onTouchEvent(ev);
    }
}
```

当系统检测到一个长按动作， `DismissOverlayView` 会显示一个退出按钮。如果用户点击它，那么我们的activity会被终止。

发送并同步数据

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/index.html

可穿戴数据层API(The Wearable Data Layer API)，Google Play services 的一部分，为手持与可穿戴应用提供了一个交流通道。此API包括一系列的数据对象，其可由系统通过网络和能告知应用数据层重要事件的监听器发送并同步：

Data Items

DataItem提供了手持设备与可穿戴设备间的自动同步的数据储存。

Messages

MessageApi类可以发送消息和善于处理远程过程调用协议（RPC），比如，从可穿戴设备上控制手持设备的媒体播放器，或在可穿戴设备上启动一个来自手持设备的intent。消息还适合单向请求或者请求/响应通信模型。如果手持设备与可穿戴设备成功连接，那么系统会将传递的消息放进队列并返回一个成功的结果码。否则，会返回一个错误。成功码并不代表成功地传递消息，这是因为设备可能在收到结果码之后断开连接。

Asset

Asset对象用于发送如图像这样的二进制数据。将资源附加到数据元，系统会自动负责传递，并通过缓存大的资源来避免重复传送以保护蓝牙带宽。

WearableListenerService (for services)

拓展的 WearableListenerService 能够监听一个service中重要的数据层事件。系统控制WearableListenerService 的生命周期，并当需要发送数据元或消息时，将其与service绑定，否则解除绑定。

DataListener (for foreground activities)

在一个前台activity中实现DataListener能够监听重要的数据通道事件。只有当用户频繁地使用应用时，用此代替WearableListenerService来监听事件变化。

Channel

使用 ChannelApi 类来从手持设备传输大的数据项到可穿戴设备，例如音乐和电影。Channel API 用于传输数据有如下的好处：

- 当使用Asset对象附加于DataItem对象时，在两个或两个以上已连接的设备间传输大的数据文件是不会自动同步。不像DataApi，Channel API 节省磁盘空间，而DataApi类是在同步已连接设备之前，就在本地设备上创建一份资源的拷贝。

- 可靠地传输对于使用MessageApi类太大的文件。
- 传输数据流，例如从网络服务器下载的音乐或者从麦克风传进来的声音。

> **Warning:** 因为这些Api是为手持设备与可穿戴设备间通信设计，所以我们只能使用这些Api来建立这些设备间的通信。例如，不能试着打开底层sockets来创建通信通道。

Android Wear支持多个可穿戴设备连接到一个手持式设备。例如，当用于在手持设备上保存了一个笔记，它会自动出现在用户的Wear设备上。为了在设备之间同步数据，Google的服务器在设备的网络上设置了一个云节点。系统将数据同步到直连的设备、云节点和通过Wi-Fi连接到云节点的可穿戴设备。

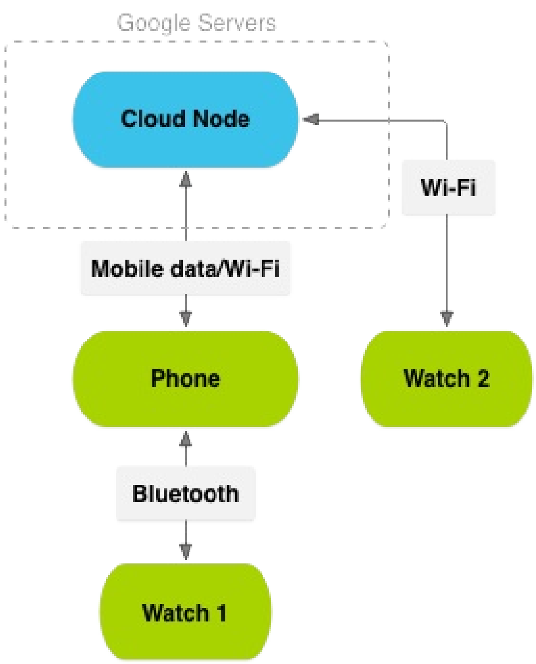


Figure 1. 一个包含手持和可穿戴设备节点的实例网络

Lessons

访问可穿戴数据层

这节课展示了如何创建一个客户端来访问数据层API。

同步数据单元

数据元是存储在一个复制而来的数据仓库中的对象，该仓库可自动由手持设备同步到可穿戴设备。

传输资源

Asset是典型地用来传输图像和媒体二进制数据。

发送与接收消息

消息被设计为自动跟踪的消息，可以在手持与可穿戴设备间来回传送。

处理数据层的事件

获知数据层的变化与事件。

访问可穿戴数据层

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/accessing.html

调用数据层API，需创建一个 GoogleApiClient 实例，所有 Google Play services APIs的主要入口点。

GoogleApiClient 提供了一个易于创建客户端实例的builder。最简单的GoogleApiClient如下：

> **Note:** 目前，此小client仅足以能启动。但是，更多创建GoogleApiClient，实现回调方法和处理错误等内容，详见 Accessing Google Play services APIs。

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(new ConnectionCallbacks() {
                @Override
                public void onConnected(Bundle connectionHint) {
                    Log.d(TAG, "onConnected: " + connectionHint);
                    // Now you can use the Data Layer API
                }
                @Override
                public void onConnectionSuspended(int cause) {
                    Log.d(TAG, "onConnectionSuspended: " + cause);
                }
        })
        .addOnConnectionFailedListener(new OnConnectionFailedListener() {
                @Override
                public void onConnectionFailed(ConnectionResult result) {
                    Log.d(TAG, "onConnectionFailed: " + result);
                }
            })
        // Request access only to the Wearable API
        .addApi(Wearable.API)
        .build();
```

> **Important:** 如果我们添加多个API到一个GoogleApiClient，那么可能会在没有安装 Android Wear app 的设备上遇到连接错误。为了连接错误，调用addApiIfAvailable()方法，并以Wearable API为参数传进该方法，从而表明client应该处理缺失的API。更多的信息，请见 Access the Wearable API.

在使用数据层API之前，通过调用connect())方法进行连接，如 Start a Connection 中所述。当系统为我们的客户端调用了onConnected()) 方法，我们就可以使用数据层API了。

同步数据单元

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/data-items.html

DataItem是指系统用于同步手持设备与可穿戴设备间数据的接口。一个DataItem通常包括以下几点：

- **Pyload** - 一个字节数组，我们可以用来设置任何数据，让我们的对象序列化和反序列化。Pyload的大小限制在100k之内。
- **Path** - 唯一且以前斜线开头的字符串（如："/path/to/data"）。

通常不直接实现DataItem，而是：

1. 创建一个PutdataRequest对象，指明一个字符串路径以唯一确定该 item。
2. 调用setData())方法设置Pyload。
3. 调用DataApi.putDataItem()方法，请求系统创建数据元。
4. 当请求数据元的时候，系统会返回正确实现DataItem接口的对象。

然而，我们建议使用Data Map来显示装在一个易用的类似Bundle接口中的数据元，而不是用setData()来处理原始字节。

用 Data Map 同步数据

使用DataMap类，将数据元处理为 Android Bundle的形式，因此会完成对象的序列化和反序列化，我们就可以以键值对（key-value）的形式操纵数据。

如何使用data map：

1. 创建一个 PutDataMapRequest对象，设置数据元的路径。
   > **Note:** 数据元的路径字符串是唯一确定的，这样能够使我们从连接任意一端访问数据元。路径须以前斜线开始。如果我们想在应用中使用分层数据，就要创建一个适合数据结构的路径方案。
2. 调用PutDataMapRequest.getDataMap())获取一个我们可以使用的data map 对象。
3. 使用put...()方法，如：putString()，为data map设置数据。
4. 调用PutDataMapRequest.asPutDataRequest())获得PutDataRequest对象。
5. 调用 DataApi.putDataItem() 请求系统创建数据元。
   > **Note:** 如果手机和可穿戴设备没有连接，数据会缓冲并在重新建立连接时同步。

接下的例子中的 `increaseCounter()` 方法展示了如何创建一个data map，并设置数据：

```
public class MainActivity extends Activity implements
        DataApi.DataListener,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener {

    private static final String COUNT_KEY = "com.example.key.count";

    private GoogleApiClient mGoogleApiClient;
    private int count = 0;

    ...

    // Create a data map and put data in it
    private void increaseCounter() {
        PutDataMapRequest putDataMapReq = PutDataMapRequest.create("/count");
        putDataMapReq.getDataMap().putInt(COUNT_KEY, count++);
        PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
        PendingResult<DataApi.DataItemResult> pendingResult =
                Wearable.DataApi.putDataItem(mGoogleApiClient, putDataReq);
    }

    ...
}
```

有关控制 PendingResult 对象的更多信息，请参见 Wait for the Status of Data Layer Calls 。

监听数据元事件

如果数据层连接的一端数据发生改变，我们很可能想要被告知在连接的另一端发生的任何改变。我们可以通过实现一个数据元事件的监听器来完成。

当定义在上一个例子中的counter的值发生改变时，下面例子的代码片段能够通知我们的app。

```
public class MainActivity extends Activity implements
        DataApi.DataListener,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener {

    private static final String COUNT_KEY = "com.example.key.count";

    private GoogleApiClient mGoogleApiClient;
    private int count = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onResume() {
        super.onStart();
        mGoogleApiClient.connect();
    }

    @Override
    public void onConnected(Bundle bundle) {
        Wearable.DataApi.addListener(mGoogleApiClient, this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        Wearable.DataApi.removeListener(mGoogleApiClient, this);
        mGoogleApiClient.disconnect();
    }

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        for (DataEvent event : dataEvents) {
            if (event.getType() == DataEvent.TYPE_CHANGED) {
                // DataItem 改变了
                DataItem item = event.getDataItem();
                if (item.getUri().getPath().compareTo("/count") == 0) {
                    DataMap dataMap = DataMapItem.fromDataItem(item).getDataMap();
                    updateCount(dataMap.getInt(COUNT_KEY));
                }
            } else if (event.getType() == DataEvent.TYPE_DELETED) {
                // DataItem 删除了
            }
        }
    }

    // 我们的更新 count 的方法
    private void updateCount(int c) { ... }

    ...
}
```

这个activity是实现了 DataItem.DataListener 接口。该activity在 `onConnected()` 方法中增加自身成为数据元事件的监听器，并在 `onPause()` 方法中移除监听器。

我们也可以用一个service实现监听，请见 监听数据层事件。

传输资源

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/assets.html

为了通过蓝牙发送大量的二进制数据，比如图片，要将一个Asset附加到数据元上，并放入复制而来的数据库中。

Assets 能够自动地处理数据缓存以避免重复发送，保护蓝牙带宽。一般的模式是：手持设备下载图像，将图片压缩到适合在可穿戴设备上显示的大小，并以Asset传给可穿戴设备。下面的例子演示此模式。

> **Note:** 尽管数据元的大小限制在100KB，但资源可以任意大。然而，传输大量资源会多方面地影响用户体验，因此，当传输大量资源时，要测试我们的应用以保证它有良好的用户体验。

传输资源

在Asset类中使用 `creat..()` 方法创建资源。下面，我们将一个bitmap转化为字节流，然后调用creatFromBytes())方法创建资源。

```
private static Asset createAssetFromBitmap(Bitmap bitmap) {
    final ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
    return Asset.createFromBytes(byteStream.toByteArray());
}
```

创建资源后，使用 DataMap 或者 PutDataRepuest 类中的 `putAsset()` 方法将其附加到数据元上，然后用 putDataItem() 方法将数据元放入数据库。

使用 PutDataRequest

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.image);
Asset asset = createAssetFromBitmap(bitmap);
PutDataRequest request = PutDataRequest.create("/image");
request.putAsset("profileImage", asset);
Wearable.DataApi.putDataItem(mGoogleApiClient, request);
```

使用 PutDataMapRequest

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.image);
Asset asset = createAssetFromBitmap(bitmap);
PutDataMapRequest dataMap = PutDataMapRequest.create("/image");
dataMap.getDataMap().putAsset("profileImage", asset)
PutDataRequest request = dataMap.asPutDataRequest();
PendingResult<DataApi.DataItemResult> pendingResult = Wearable.DataApi
        .putDataItem(mGoogleApiClient, request);
```

接收资源

创建资源后，我们可能需要在另一连接端读取资源。以下是如何实现回调以发现资源变化和提取Asset对象。

```
@Override
public void onDataChanged(DataEventBuffer dataEvents) {
  for (DataEvent event : dataEvents) {
    if (event.getType() == DataEvent.TYPE_CHANGED &&
        event.getDataItem().getUri().getPath().equals("/image")) {
      DataMapItem dataMapItem = DataMapItem.fromDataItem(event.getDataItem());
      Asset profileAsset = dataMapItem.getDataMap().getAsset("profileImage");
      Bitmap bitmap = loadBitmapFromAsset(profileAsset);
      // Do something with the bitmap
    }
  }
}

public Bitmap loadBitmapFromAsset(Asset asset) {
    if (asset == null) {
        throw new IllegalArgumentException("Asset must be non-null");
    }
    ConnectionResult result =
           mGoogleApiClient.blockingConnect(TIMEOUT_MS, TimeUnit.MILLISECONDS);
    if (!result.isSuccess()) {
        return null;
    }
    // convert asset into a file descriptor and block until it's ready
    InputStream assetInputStream = Wearable.DataApi.getFdForAsset(
            mGoogleApiClient, asset).await().getInputStream();
            mGoogleApiClient.disconnect();

    if (assetInputStream == null) {
        Log.w(TAG, "Requested an unknown Asset.");
        return null;
    }
    // decode the stream into a bitmap
    return BitmapFactory.decodeStream(assetInputStream);
}
```

发送与接收消息

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/messages.html

使用MessageApi发送消息，要附加以下几项：

- 任一payload（可选）
- 唯一标识消息动作的路径

不像数据元，Messages（消息）在手持和可穿戴应用之间没有同步。Messages是单向交流机制，这有利于远程进程调用(RPC)，比如：发送消息到可穿戴设备以开启activity。

多个可穿戴设备可以连接到一台用户的手持设备。在网络中每个已连接的设备被视为一个节点（*node*）。由于有多个已连接的设备，我们必须考虑哪个节点收到消息。例如，在一个在可穿戴设备上接收语音数据的语音转录应用中，我们应该发送消息到一个具有处理能力和电池容量的节点来处理请求，例如一个手持式设备。

> **Note:** Google Play services 7.3.0版之前，一次只有一个可穿戴设备可以连接到手持设备。我们需要将现有的代码升级，以考虑到多个连接节点的功能。如果我们不作出修改，那么我们的消息可能不会传到想要的设备。

发送消息

一个可穿戴应用可以为用户提供如语音转录等功能。用户可以对着他们可穿戴设备的麦克风说话，然后就会将语音保存成一个笔记。由于一个可穿戴设备通常没有足够的处理能力和电池容量来处理语音转录activity，所以应用应该将这个工作留给一个更加有能力的、已连接的设备来处理。

下面几个小节介绍如何通知那些可以处理activity请求的设备节点，发现有能力满足请求的节点，并发送消息给那些节点。

通知节点功能

使用 MessageApi 类发送请求，来从一个可穿戴设备启动一个手持设备的activity。由于一个手持式设备可以连接多个可穿戴设备，所以可穿戴应用需要确定一个已连接的节点是否有能力启动activity。在我们的手持式应用中，通知其它节点：我们的手持式应用所在的节点提供了上述指定的功能。

为了把我们的手持式应用的功能通知其它节点，需要：

1. 在工程的 `res/values/` 目录下创建一个名为 `wear.xml` 的 XML 文件。
2. 在 `wear.xml` 文件中添加一个名为 `android_wear_capabilities` 的资源。
3. 定义设备可以提供的功能。

> **Note:** 功能是我们自定义的字符串，它在我们的应用中必须是唯一的。

下面这个例子介绍了如何将一个名为 `voice_transcription` 的功能添加到 `wear.xml` 中：

```
<resources>
    <string-array name="android_wear_capabilities">
        <item>voice_transcription</item>
    </string-array>
</resources>
```

检索具有相关功能的节点

首先，我们可以通过调用 CapabilityApi.getCapability() 方法来检测具有相关功能的节点。下面的例子介绍了如何手动检索具有 `voice_transcription` 功能的节点：

```
private static final String
        VOICE_TRANSCRIPTION_CAPABILITY_NAME = "voice_transcription";

private GoogleApiClient mGoogleApiClient;

...

private void setupVoiceTranscription() {
    CapabilityApi.GetCapabilityResult result =
            Wearable.CapabilityApi.getCapability(
                    mGoogleApiClient, VOICE_TRANSCRIPTION_CAPABILITY_NAME,
                    CapabilityApi.FILTER_REACHABLE).await();

    updateTranscriptionCapability(result.getCapability());
}
```

为了在连接到可穿戴设备的时候检测有能力的节点，注册一个 CapabilityApi.CapabilityListener() 实例到 GoogleApiClient。下面的例子介绍了如何注册该监听器和检索具有 `voice_transcription` 功能的节点。

```java
private void setupVoiceTranscription() {
    ...

    CapabilityApi.CapabilityListener capabilityListener =
            new CapabilityApi.CapabilityListener() {
                @Override
                public void onCapabilityChanged(CapabilityInfo capabilityInfo) {
                    updateTranscriptionCapability(capabilityInfo);
                }
            };

    Wearable.CapabilityApi.addCapabilityListener(
            mGoogleApiClient,
            capabilityListener,
            VOICE_TRANSCRIPTION_CAPABILITY_NAME);
}
```

> **Note:** 如果我们创建一个继承 WearableListenerService 的 service 来检测功能的变化，我们可能要重写 onConnectedNodes()) 方法来监听细微的连接细节，例如，一个可穿戴设备与手持式设备从Wi-Fi连接切换到蓝牙连接。关于一个实现的例子，请查看在 FindMyPhone 示例中的 `DisconnectListenerService` 类。更多关于如何监听重要事件的内容，请见监听数据层事件。

检测到有能力的节点之后，需要确定将消息发送到哪里。我们需要选择与可穿戴设备邻近的节点，这样可以最小化多个节点间的消息路由。一个邻近的节点被定义为一个直接与设备连接的节点。调用 Node.isNearby()) 来确定一个节点是否是邻近的。

下面的例子介绍了如何确定最佳节点：

```java
private String transcriptionNodeId = null;

private void updateTranscriptionCapability(CapabilityInfo capabilityInfo) {
    Set<Node> connectedNodes = capabilityInfo.getNodes();

    transcriptionNodeId = pickBestNodeId(connectedNodes);
}

private String pickBestNodeId(Set<Node> nodes) {
    String bestNodeId = null;
    // Find a nearby node or pick one arbitrarily
    for (Node node : nodes) {
        if (node.isNearby()) {
            return node.getId();
         }
         bestNodeId = node.getId();
    }
    return bestNodeId;
}
```

传送消息

一旦我们确定了最佳节点，使用 MessageApi 发送消息。

下面的例子介绍了如何从一个可穿戴设备发送消息到具有语音转录功能的节点。在我们试图发送消息之前，需要判断节点是否可用。这个调用是同步的，它在系统将传送的消息放到队列前会一直阻塞。

> **Note:** 一个成功结果码并不保证消息是否传送成功。如果我们的应用需要数据的可靠性，那么使用 DataItem 对象或者 ChannelApi 类在设备间发送数据。

```
public static final String VOICE_TRANSCRIPTION_MESSAGE_PATH = "/voice_transcription";

private void requestTranscription(byte[] voiceData) {
    if (transcriptionNodeId != null) {
        Wearable.MessageApi.sendMessage(googleApiClient, transcriptionNodeId,
            VOICE_TRANSCRIPTION_MESSAGE_PATH, voiceData).setResultCallback(
                  new ResultCallback() {
                      @Override
                      public void onResult(SendMessageResult sendMessageResult) {
                          if (!sendMessageResult.getStatus().isSuccess()) {
                              // Failed to send message
                          }
                      }
                  }
            );
    } else {
        // Unable to retrieve node with transcription capability
    }
}
```

> **Note:** 阅读 Communicate with Google Play Services 了解更多关于异步和同步调用，以及何时使用哪个。

我们还可以广播消息给所有已连接的节点。为了获得我们可以发送消息的已连接节点，需要实现下面的代码：

```
private Collection<String> getNodes() {
    HashSet <String>results = new HashSet<String>();
    NodeApi.GetConnectedNodesResult nodes =
            Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).await();
    for (Node node : nodes.getNodes()) {
        results.add(node.getId());
    }
    return results;
}
```

接收消息

为了在收到消息时被提醒，我们可以实现 MessageListener 接口来提供消息事件的监听。然后，我们需要在 MessageApi.addListener() 方法中注册监听。这个例子展示如何通过检查 `VOICE_TRANSCRIPTION_MESSAGE_PATH` 来实现监听器。如果该条件是true，就会启动特定的activity来处理语音数据。

```
@Override
public void onMessageReceived(MessageEvent messageEvent) {
    if (messageEvent.getPath().equals(VOICE_TRANSCRIPTION_MESSAGE_PATH)) {
        Intent startIntent = new Intent(this, MainActivity.class);
        startIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startIntent.putExtra("VOICE_DATA", messageEvent.getData());
        startActivity(startIntent);
    }
}
```

这仅是实现更多细节的一小段。关于如何在 service 或 activity 实现完整的监听，请参见 监听数据传输层事件 。

处理数据层的事件

> 编写:wly2014 - 原文: http://developer.android.com/training/wearables/data-layer/events.html

当做出数据层上的调用时，我们可以得到它完成后的调用状态，也可以用监听器监听到调用最终实现的改变。

等待数据层调用的状态

注意到，调用数据层API，有时会返回 PendingResult，如 putDataItem()。PendingResult 一被创建，操作就会在后台排列等候。之后我们若无动作，这些操作最终会默默完成。然而，通常要处理操作完成后的结果，PendingResult 能够让我们同步或异步地等待结果。

异步调用

若代码运行在主UI线程上，不要让数据层API调用阻塞UI。我们可以增加一个回调到 PendingResult 对象来运行异步调用，该回调函数将在操作完成时触发。

```
pendingResult.setResultCallback(new ResultCallback<DataItemResult>() {
    @Override
    public void onResult(final DataItemResult result) {
        if(result.getStatus().isSuccess()) {
            Log.d(TAG, "Data item set: " + result.getDataItem().getUri());
        }
    }
});
```

同步调用

如果代码是运行在后台服务的一个独立的处理线程上（WearableListenerService的情况），则调用导致的阻塞没影响。在这种情况下,我们可以用 PendingResult对象调用await())，它将阻塞至请求完成,并返回一个Result对象：

```
DataItemResult result = pendingResult.await();
if(result.getStatus().isSuccess()) {
    Log.d(TAG, "Data item set: " + result.getDataItem().getUri());
}
```

监听数据层事件

因为数据层在手持和可穿戴设备间同步并发送数据，所以通常要监听重要事件，例如创建数据元，接收消息，或连接可穿戴设备和手机。

对于监听数据层事件，有两种选择：

- 创建一个继承自 WearableListenerService 的 service。
- 创建一个实现 DataApi.DataListener 接口的 activity。

通过这两种选择，为我们感兴趣的事件重写数据事件回调方法。

使用 WearableListenerService

通常，我们在手持设备和可穿戴设备上都创建该 service 的实例。如果我们不关心其中一个应用中的数据事件，就不需要在相应的应用中实现此 service。

例如，我们可以在一个手持设备应用程序上操作数据元对象，可穿戴设备应用监听这些更新来更新自身的UI。而可穿戴不更新任何数据元，所以手持设备应用不监听任何可穿戴式设备应用的数据事件。

我们可以用 WearableListenerService 监听如下事件：

- onDataChanged()) - 当数据元对象创建，更改，删除时调用。一连接端的事件将触发两端的回调方法。
- onMessageReceived()) - 消息从一连接端发出，在另一连接端触发此回调方法。
- onPeerConnected()) 和 onPeerDisconnected()) - 当与手持或可穿戴设备连接或断开时调用。一连接端连接状态的改变会在两端触发此回调方法。

创建WearableListenerService，我们需要：

1. 创建一个继承自 WearableListenerService 的类。
2. 监听我们关心的事件，比如 onDataChanged())。
3. 在Android manifest中声明一个intent filter，把我们的 WearableListenerService 通知给系统。这样允许系统在需要时绑定我们的 service。

下例展示如何实现一个简单的 WearableListenerService：

```
public class DataLayerListenerService extends WearableListenerService {

    private static final String TAG = "DataLayerSample";
    private static final String START_ACTIVITY_PATH = "/start-activity";
    private static final String DATA_ITEM_RECEIVED_PATH = "/data-item-received";

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "onDataChanged: " + dataEvents);
        }
        final List events = FreezableUtils
                .freezeIterable(dataEvents);

        GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .build();

        ConnectionResult connectionResult =
                googleApiClient.blockingConnect(30, TimeUnit.SECONDS);

        if (!connectionResult.isSuccess()) {
            Log.e(TAG, "Failed to connect to GoogleApiClient.");
            return;
        }

        // Loop through the events and send a message
        // to the node that created the data item.
        for (DataEvent event : events) {
            Uri uri = event.getDataItem().getUri();

            // Get the node id from the host value of the URI
            String nodeId = uri.getHost();
            // Set the data of the message to be the bytes of the URI
            byte[] payload = uri.toString().getBytes();

            // Send the RPC
            Wearable.MessageApi.sendMessage(googleApiClient, nodeId,
                    DATA_ITEM_RECEIVED_PATH, payload);
        }
    }
}
```

这是Android mainfest中相应的intent filter：

```
<service android:name=".DataLayerListenerService">
  <intent-filter>
      <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
  </intent-filter>
</service>
```

数据层回调权限

为了在数据层事件上向我们的应用传送回调方法，Google Play services 绑定到我们的 WearableListenerService，并通过IPC调用回调方法。这样的结果是，我们的回调方法继承了调用进程的权限。

如果我们想在一个回调中执行权限操作，安全检查会失败，因为回调是以调用进程的身份运行，而不是应用程序进程的身份运行。

为了解决这个问题，在进入IPC后使用 clearCallingIdentity()) 重置身份，当完成权限操作后，使用 restoreCallingIdentity()) 恢复身份:

```
long token = Binder.clearCallingIdentity();
try {
    performOperationRequiringPermissions();
} finally {
    Binder.restoreCallingIdentity(token);
}
```

使用一个Listener Activity

如果我们的应用只关心当用户与应用交互时产生的数据层事件，并且不需要一个长时间运行的 service 来处理每一次数据的改变，那么我们可以在一个 activity 中通过实现如下一个和多个接口来监听事件：

- DataApi.DataListener
- MessageApi.MessageListener
- NodeApi.NodeListener

创建一个 activity 监听数据事件，需要：

1. 实现所需的接口。
2. 在 onCreate(Bundle)) 中创建 GoogleApiClient 实例。
3. 在 onStart()) 中调用 connect()) 将客户端连接到 Google Play services。
4. 当连接到Google Play services后，系统调用 onConnected())。这里是我们调用 DataApi.addListener()，MessageApi.addListener() 或 NodeApi.addListener()，以告知 Google Play services 我们的 activity 要监听数据层事件的地方。
5. 在 onStop()) 中，用 DataApi.removeListener()), MessageApi.removeListener()) 或NodeApi.removeListener()) 注销监听。
6. 基于我们实现的接口继而实现 onDataChanged(), onMessageReceived(), onPeerConnected()和 onPeerDisconnected()。

这是实现DataApi.DataListener的例子 ：

```
public class MainActivity extends Activity implements
        DataApi.DataListener, ConnectionCallbacks, OnConnectionFailedListener {

    private GoogleApiClient mGoogleApiClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (!mResolvingError) {
            mGoogleApiClient.connect();
        }
    }

    @Override
    public void onConnected(Bundle connectionHint) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Connected to Google Api Service");
        }
        Wearable.DataApi.addListener(mGoogleApiClient, this);
    }

    @Override
    protected void onStop() {
        if (null != mGoogleApiClient && mGoogleApiClient.isConnected()) {
            Wearable.DataApi.removeListener(mGoogleApiClient, this);
            mGoogleApiClient.disconnect();
        }
        super.onStop();
    }

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        for (DataEvent event : dataEvents) {
            if (event.getType() == DataEvent.TYPE_DELETED) {
                Log.d(TAG, "DataItem deleted: " + event.getDataItem().getUri());
            } else if (event.getType() == DataEvent.TYPE_CHANGED) {
                Log.d(TAG, "DataItem changed: " + event.getDataItem().getUri());
            }
        }
    }
```

```
}
```

创建表盘

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/index.html

Android Wear 的表盘是一个动态的数字画布，它用颜色、动画和相关的上下文信息来表示时间。Android Wear companion app 提供了不同风格和形状的表盘。当用户选择可穿戴设备应用或者配套应用上可用的表盘，可穿戴设备会提供表盘的预览并让用户设置选项。

Android Wear 允许我们为 Wear 设备创建自定义的表盘。当用户安装一个包含表盘的可穿戴应用的手持式应用时，它们可以在手持式设备上的 Android Wear 配套应用和在可穿戴设备上的表盘选择器中使用。

这个课程教我们实现自定义表盘并将它们打包进一个可穿戴应用。这节课还覆盖设计方面的考虑和实现提示，从而确保我们的设计整合到系统 UI 并且节能。

> **Note:** 我们推荐使用 Android Studio 做 Android Wear 开发，它提供工程初始配置，库包含和方便的打包流程，这些在ADT中是没有的。这系列教程假定你正在使用Android Studio。

Lesson

设计表盘

学习如何设计一个可以工作在 Android Wear 设备上的表盘。

构建表盘服务

学习如何在表盘的生命周期期间响应重要的时间。

绘制表盘

学习如何在一个 Wear 设备的屏幕上绘制表盘。

在表盘上显示信息

学习如何将上下文信息集成到表盘中。

提供配置 Activity

学习如何创建带有可配置参数的表盘。

定位常见的问题

学习如何在开发表盘的时候修改常见的问题。

优化性能和电池使用时间

学习如何提高动画的帧速率和节能。

设计表盘

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/designing.html

类似于设计传统的表盘，创建 Android Wear 的表盘是一个清晰地显示时间的练习。Android Wear 设备为表盘提供了高级的功能，我们可以运用这些功能到我们的设计当中，例如鲜艳的色彩、动态的背景、动画和数据整合。然而，我们必须考虑到很多其它设计上的因素。

这节课总结了设计表盘需要考虑的因素和通用准则。更多关于这方面的内容，请见 Watch Faces for Android Wear 设计指引。

遵守设计准侧

当我们设计表盘的外观和表盘需要向用户表达哪些类型的信息的时候，请考虑一下这些设计准侧：

为方形和圆形的设备作出规划

我们的设计应该可以运行在方形和圆形的 Android Wear 设备上，包括那些使用感知形状的 Layout 的设备。

支持所有的显示模式

我们的表盘应该支持有限颜色的环境模式（ambient mode）和全彩色动画的交互模式（interactive mode）。

优化特殊屏幕的技术

在环境模式下，表盘应该让大部分像素保持黑色。根据屏幕技术，我们需要避免使用大块的白像素，仅仅使用黑色和白色，并禁用反锯齿。

容纳系统 *UI* 组件

我们的设计应该确保系统指示图标可见，当 notification cards 出现在屏幕上的时候用户还可以看到时间。

整合数据

我们的表盘可以利用配套手机设备上的传感器和蜂窝数据连接，来显示相关的上下文数据，例如天气或者用户的下一个日程表事件。

提供设置选项

我们可以让用户配置可穿戴应用或者 Android Wear 配套应用上某些设计特征（如颜色和尺寸）。





Figure 1. 表盘的例子.

更多关于 Android Wear 表盘的设计，请见 Watch Faces for Android Wear 设计指引。

创建实现策略

完成表盘的设计后，我们需要决定如何获得必要的数据和将表盘绘制到可穿戴设备上。大部分实现方案由如下部分组成：

- 一幅或多幅背景图片
- 接收需要数据的应用代码
- 绘制背景图片上的文本和形状的应用代码

我们一般在交互模式和环境模式使用两幅不同的背景图片。环境模式下的背景一般是全黑的。Android Wear 设备的屏幕密度（hdpi）应该是 320 x 320 像素，这样可以同时兼容方形和圆形设备。背景图片的四角在圆形设备上是不可见的。在我们的代码中，我们可以检测到设备屏幕的尺寸。如果设备的分辨率比图片的低，那么按比例缩小背景图片。为了提高性能，我们应该只对背景图片缩放一次并保存缩放后的 bitmap。

我们应该在需要时运行代码来检索上下文数据和保存结果，使得在每次绘制表盘的时候重用数据。例如，我们不需要每隔一分钟去刷新一次天气。

为了增加电池使用时间，在环境模式绘制表盘的应用代码应该相对简单。在环境模式下，我们通常用一组有限的颜色来绘制形状的轮廓。在交互模式下，我们可以使用全色彩、复杂的形状、渐变和动画来绘制表盘。

后面的课程将会介绍如何详细地实现表盘。

构建表盘服务

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/service.html

Android Wear 的表盘在可穿戴应用中实现为服务（services）和包。当用户安装一个包含表盘的可穿戴应用的手持式应用时，这些表盘在手持式设备的 Android Wear 配套应用和可穿戴表盘选择器中可用。当用户选择一个可用的表盘时，可穿戴设备会显示表盘并且按需要调用它的服务毁掉方法。

这节课介绍如何配置包含表盘的 Android 工程和如何实现表盘服务。

创建并配置工程

在 Android Studio 中为表盘创建一个 Android 工程，需要：

1. 打开 Android Studio。
2. 创建一个新的工程：
    - 如果没有打开过任何工程，那么在 **Welcome** 界面中点击 **New Project**。
    - 如果已经打开过工程，那么在 **File** 菜单中选择 **New Project**。
3. 填写应用名字，然后点击 **Next**。
4. 选择 **Phone and Tablet** 尺寸系数。
5. 在 **Minimum SDK** 下拉菜单选择 API 18。
6. 选择 **Wear** 尺寸系数。
7. 在 **Minimum SDK** 下拉菜单选择 API 21，然后点击 **Next**。
8. 选择 **Add No Activity** 然后在接下来的两个界面点击 **Next** 。
9. 点击 **Finish**。
10. 在 IDE 窗口点击 **View > Tool Windows > Project**。

至此，Android Studio 创建了一个含有 `wear` 和 `mobile` 模块的工程。更多关于创建工程的内容，请见 Creating a Project。

依赖

Wearable Support 库提供了必要的类，我们可以继承这些类来创建表盘的实现。需要用 Google Play services client 库（`play-services` 和 `play-services-wearable`）在配套设备和含有可穿戴数据层 API 的可穿戴应用之间同步数据项。

当我们按照上述的方法创建工程时，Android Studio 会自动添加需要的条目到 `build.gradle` 文件。

Wearable Support 库 API 参考资源

该参考文档提供了用于实现表盘的详细信息。详见 API 参考文档。

在 Eclipse ADT 中下载 Wearable Support 库

如果你使用 Eclipse ADT，那么请下载 Wearable Support 库 并且将该库作为依赖包含在你的工程当中。

声明权限

表盘需要 `PROVIDE_BACKGROUND` 和 `WAKE_LOCK` 权限。在可穿戴和手持式应用的 manifest 文件中 `manifest` 节点下添加如下权限：

```
<manifest ...>
    <uses-permission
        android:name="com.google.android.permission.PROVIDE_BACKGROUND" />
    <uses-permission
        android:name="android.permission.WAKE_LOCK" />
    ...
</manifest>
```

> **Caution:** 手持式应用必须包括所有在可穿戴应用中声明的权限。

实现服务和回调方法

Android Wear 的表盘实现为服务(services)。当表盘处于活动状态时，系统会在时间改变或者出现重要的时间（如切换到环境模式或者接收到一个新的通知）的时候调用服务的方法。服务实现接着根据更新的时间和其它相关的数据将表盘绘制到屏幕上。

实现一个表盘，我们需要继承 `CanvasWatchFaceService` 和 `CanvasWatchFaceService.Engine` 类，然后重写 `CanvasWatchFaceService.Engine` 类的回调方法。这些类都包含在 Wearable Support 库里。

下面的代码片段略述了我们需要实现的主要方法：

```
public class AnalogWatchFaceService extends CanvasWatchFaceService {

    @Override
    public Engine onCreateEngine() {
        /* provide your watch face implementation */
        return new Engine();
    }

    /* implement service callback methods */
    private class Engine extends CanvasWatchFaceService.Engine {

        @Override
        public void onCreate(SurfaceHolder holder) {
            super.onCreate(holder);
            /* initialize your watch face */
        }

        @Override
        public void onPropertiesChanged(Bundle properties) {
            super.onPropertiesChanged(properties);
            /* get device features (burn-in, low-bit ambient) */
        }

        @Override
        public void onTimeTick() {
            super.onTimeTick();
            /* the time changed */
        }

        @Override
        public void onAmbientModeChanged(boolean inAmbientMode) {
            super.onAmbientModeChanged(inAmbientMode);
            /* the wearable switched between modes */
        }

        @Override
        public void onDraw(Canvas canvas, Rect bounds) {
            /* draw your watch face */
        }

        @Override
        public void onVisibilityChanged(boolean visible) {
            super.onVisibilityChanged(visible);
            /* the watch face became visible or invisible */
        }
    }
}
```

> **Note:** Android SDK 里的 *WatchFace* 示例示范了如何通过继承 `CanvasWatchFaceService` 类来实现模拟和数字表盘。这个示例位于 `android-sdk/samples/android-21/wearable/WatchFace` 目录。

`CanvasWatchFaceService` 类提供一个类似 View.invalidate()) 方法的销毁机制。当我们想要系统重新绘制表盘时，我们可以在实现中调用 `invalidate()` 方法。在主 UI 线程中，我们可以只用 `invalidate()` 方法。然后调用 `postInvalidate()` 方法从其它的的线程中销毁画布。

更多关于实现 `CanvasWatchFaceService.Engine` 类的方法，请见绘制表盘。

注册表盘服务

实现完表盘服务之后，我们需要在可穿戴应用的 manifest 文件中注册该实现。当用户安装此应用时，系统会使用关于服务的信息，使得可穿戴设备上 Android Wear 配套应用和表盘选择器里的表盘可用。

下面的代码片段介绍了如何在 application 节点下注册一个表盘实现：

```
<service
    android:name=".AnalogWatchFaceService"
    android:label="@string/analog_name"
    android:allowEmbedded="true"
    android:taskAffinity=""
    android:permission="android.permission.BIND_WALLPAPER" >
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/watch_face" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview"
        android:resource="@drawable/preview_analog" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview_circular"
        android:resource="@drawable/preview_analog_circular" />
    <intent-filter>
        <action android:name="android.service.wallpaper.WallpaperService" />
        <category
            android:name=
            "com.google.android.wearable.watchface.category.WATCH_FACE" />
    </intent-filter>
</service>
```

当向用户展示所有安装在可穿戴设备的表盘时，设备上的Android Wear 配套应用和表盘选择器使用 `com.google.android.wearable.watchface.preview` 元数据项定义的预览图。为了取得这个 drawable，可以运行 Android Wear 设备或者模拟器上的表盘并截图。在 hdpi 屏幕的 Android Wear 设备上，预览图像一般是 320x320 像素。

圆形设备上看起来非常不同的表盘可以提供圆形和方形的预览图。使用 `com.google.android.wearable.watchface.preview` 元数据项指定一个圆形的预览图。如果一个表盘包含两种预览图，可穿戴应用上的配套应用和表盘选择器会根据手表的形状选择适合的

预览图。如果没有包含圆形的预览图，那么方形和圆形的设备都会用方形的预览图。对于圆形的设备，方形的预览图会被一个圆形剪裁掉。

`android.service.wallpaper` 元数据项指定包含 `wallpaper` 节点的 `watch_face.xml` 资源文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android" />
```

我们的可穿戴应用可以包含多个表盘。我们必须为每个表盘实现添加一个服务节点到可穿戴应用的 manifest 文件中。

绘制表盘

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/drawing.html

配置完工程和添加了实现表盘服务（watch face service）的类之后，我们可以开始编写初始化和绘制自定义表盘的代码了。

这节课通过 Android SDK 中的 *WatchFace* 示例，来介绍系统是如何调用表盘服务的方法。这个示例位于 `android-sdk/samples/android-21/wearable/WatchFace` 目录。这里描述服务实现的很多方面（例如初始化和检测设备功能）可以应用到任意表盘，所以我们可以重用一些代码到我们的表盘当中。

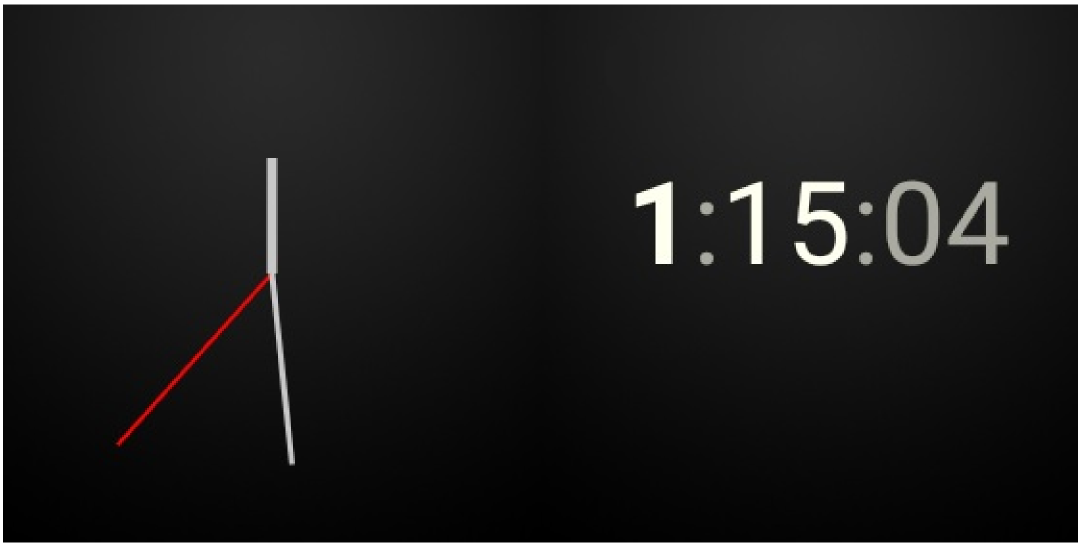


Figure 1. *WatchFace* 示例中的模拟和数字表盘

初始化表盘

当系统加载我们的服务时，我们应该分配和初始化表盘需要的大部分资源，包括加载位图资源、创建定时器对象来运行自定义动画、配置颜色风格和执行其他运算。我们通常只执行一次这些操作和重用它们的结果。这个习惯可以提高表盘的性能并且更容易地维护代码。

初始化表盘，需要：

1. 为自定义定时器、图形对象和其它组件声明变量。
2. 在 `Engine.onCreate()` 方法中初始化表盘组件。
3. 在 `Engine.onVisibilityChanged()` 方法中初始化自定义定时器。

下面的部分详细介绍了上述几个步骤。

声明变量

当系统加载我们的服务时，我们初始化的那些资源需要在我们实现的不同点都可以被访问，所以我们可以重用这些资源。我们可以通过在 `WatchFaceService.Engine` 实现中为这些资源声明成员变量来达到上述目的。

为下面的组件声明变量：

图形对象

大部分表盘至少包含一个位图用于表盘的背景，如创建实施策略描述的一样。我们可以使用额外的位图图像来表示表盘的时钟指针或者其它设计元素。

定时计时器

当时间变化时，系统每隔一分钟会通知表盘一次，但一些表盘会根据自定义的时间间隔来运行动画。在这种情况下，我们需要用一个按照所需频率计数的自定义定时器来刷新表盘。

时区变化接收器

用户可以在旅游的时候调整时区，系统会广播这个事件。我们的服务实现必须注册一个广播接收器，该广播接收器用于接收时区改变或者更新时间的通知。

WatchFace 示例中的 `AnalogWatchFaceService.Engine` 类定义了上述变量（见下面的代码）。自定义定时器实现为一个 Handler 实例，该 Handler 实例使用线程的消息队列发送和处理延迟的消息。对于这个特定的表盘，自定义定时器每秒计数一次。当定时器计数，handler 调用 `invalidate()` 方法，然后系统调用 `onDraw()` 方法重新绘制表盘。

```
private class Engine extends CanvasWatchFaceService.Engine {
    static final int MSG_UPDATE_TIME = 0;

    /* a time object */
    Time mTime;

    /* device features */
    boolean mLowBitAmbient;

    /* graphic objects */
    Bitmap mBackgroundBitmap;
    Bitmap mBackgroundScaledBitmap;
    Paint mHourPaint;
    Paint mMinutePaint;
    ...

    /* handler to update the time once a second in interactive mode */
    final Handler mUpdateTimeHandler = new Handler() {
        @Override
        public void handleMessage(Message message) {
            switch (message.what) {
                case MSG_UPDATE_TIME:
                    invalidate();
                    if (shouldTimerBeRunning()) {
                        long timeMs = System.currentTimeMillis();
                        long delayMs = INTERACTIVE_UPDATE_RATE_MS
                                - (timeMs % INTERACTIVE_UPDATE_RATE_MS);
                        mUpdateTimeHandler
                            .sendEmptyMessageDelayed(MSG_UPDATE_TIME, delayMs);
                    }
                    break;
            }
        }
    };

    /* receiver to update the time zone */
    final BroadcastReceiver mTimeZoneReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            mTime.clear(intent.getStringExtra("time-zone"));
            mTime.setToNow();
        }
    };

    /* service methods (see other sections) */
    ...
}
```

初始化表盘组件

在为位图资源、色彩风格和其它每次重新绘制表盘都会重用的组件声明成员变量之后，在系统加载服务时初始化这些组件。只初始化这些组件一次，然后重用它们以提升性能和电池使用时间。

在 `Engine.onCreate()` 方法中，初始化下面的组件：

- 加载背景图片。
- 创建风格和色彩来绘制图形对象。
- 分配一个对象来保存时间。
- 配置系统 UI。

在 `AnalogWatchFaceService` 类的 `Engine.onCreate()` 方法初始化这些组件的代码如下：

```
@Override
public void onCreate(SurfaceHolder holder) {
    super.onCreate(holder);

    /* configure the system UI (see next section) */
    ...

    /* load the background image */
    Resources resources = AnalogWatchFaceService.this.getResources();
    Drawable backgroundDrawable = resources.getDrawable(R.drawable.bg);
    mBackgroundBitmap = ((BitmapDrawable) backgroundDrawable).getBitmap();

    /* create graphic styles */
    mHourPaint = new Paint();
    mHourPaint.setARGB(255, 200, 200, 200);
    mHourPaint.setStrokeWidth(5.0f);
    mHourPaint.setAntiAlias(true);
    mHourPaint.setStrokeCap(Paint.Cap.ROUND);
    ...

    /* allocate an object to hold the time */
    mTime = new Time();
}
```

当系统初始化表盘时，只会加载背景位图一次。图形风格被 Paint 类实例化。然后我们在 `Engine.onDraw()` 方法中使用这些风格来绘制表盘的组件，如绘制表盘描述的那样。

初始化自定义定时器

作为表盘开发者，我们通过使定时器按照要求的频率计数，来决定设备在交互模式时多久更新一次表盘。这使得我们可以创建自定义的动画和其它视觉效果。

> **Note:** 在环境模式下，系统不会可靠地调用自定义定时器。关于在环境模式下更新表盘的内容，请看在环境模式下更新表盘。

在声明变量部分介绍了一个 `AnalogWatchFaceService` 类定义的每秒计数一次的定时器例子。在 `Engine.onVisibilityChanged()` 方法里，如果满足如下两个条件，则启动自定义定时器：

- 表盘可见的。
- 设备处于交互模式。

如果有必要，`AnalogWatchFaceService` 会调度下一个定时器进行计数：

```
private void updateTimer() {
    mUpdateTimeHandler.removeMessages(MSG_UPDATE_TIME);
    if (shouldTimerBeRunning()) {
        mUpdateTimeHandler.sendEmptyMessage(MSG_UPDATE_TIME);
    }
}

private boolean shouldTimerBeRunning() {
    return isVisible() && !isInAmbientMode();
}
```

该自定义定时器每秒计数一次，如声明变量介绍的一样。

在 `Engine.onVisibilityChanged()` 方法中，按要求启动定时器并为时区的变化注册接收器：

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);

    if (visible) {
        registerReceiver();

        // Update time zone in case it changed while we weren't visible.
        mTime.clear(TimeZone.getDefault().getID());
        mTime.setToNow();
    } else {
        unregisterReceiver();
    }

    // Whether the timer should be running depends on whether we're visible and
    // whether we're in ambient mode), so we may need to start or stop the timer
    updateTimer();
}
```

当表盘可见时，`onVisibilityChanged()` 方法为时区变化注册了接收器，并且如果设备在交互模式，则启动自定义定时器。当表盘不可见，这个方法停止自定义定时器并且注销检测时区变化的接收器。下面是 `registerReceiver()` 和 `unregisterReceiver()` 方法的实现：

```
private void registerReceiver() {
    if (mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = true;
    IntentFilter filter = new IntentFilter(Intent.ACTION_TIMEZONE_CHANGED);
    AnalogWatchFaceService.this.registerReceiver(mTimeZoneReceiver, filter);
}

private void unregisterReceiver() {
    if (!mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = false;
    AnalogWatchFaceService.this.unregisterReceiver(mTimeZoneReceiver);
}
```

在环境模式下更新表盘

在环境模式下，系统每分钟调用一次 `Engine.onTimeTick()` 方法。通常在这种模式下，每分钟更新一次表盘已经足够了。为了在环境模式下更新表盘，我们必须使用一个在初始化自定义定时器介绍的自定义定时器。

在环境模式下，大部分表盘实现在 `Engine.onTimeTick()` 方法中简单地销毁画布来重新绘制表盘：

```
@Override
public void onTimeTick() {
    super.onTimeTick();

    invalidate();
}
```

配置系统 UI

表盘不应该干涉系统 UI 组件，在 Accommodate System UI Element 中有介绍。如果我们的表盘背景比较亮或者在屏幕的底部附近显示了信息，那么我们可能要配置 notification cards 的尺寸或者启用背景保护。

当表盘在动的时候，Android Wear 允许我们配置系统 UI 的下面几个方面：

- 指定第一个 notification card 离屏幕有多远。
- 指定系统是否将时间绘制在表盘上。
- 在环境模式下，显示或者隐藏 notification card。
- 用纯色背景保护系统指针。

- 指定系统指针的位置。

为了配置这些方面的系统 UI，需要创建一个 `WatchFaceStyle` 实例并且将其传进 `Engine.setWatchFaceStyle()` 方法。

下面是 `AnalogWatchFaceService` 类配置系统 UI 的方法：

```
@Override
public void onCreate(SurfaceHolder holder) {
    super.onCreate(holder);

    /* configure the system UI */
    setWatchFaceStyle(new WatchFaceStyle.Builder(AnalogWatchFaceService.this)
            .setCardPeekMode(WatchFaceStyle.PEEK_MODE_SHORT)
            .setBackgroundVisibility(WatchFaceStyle
                                    .BACKGROUND_VISIBILITY_INTERRUPTIVE)
            .setShowSystemUiTime(false)
            .build());
    ...
}
```

上述的代码将 card 配置成一行高，card 的背景只会简单地显示和只用于中断的 notification，不会显示系统时间（因为表盘会绘制自己的时间）。

我们可以在表盘实现的任意时刻配置系统的 UI 风格。例如，如果用户选择了白色背景，我们可以为系统指针添加背景保护。

更多关于配置系统 UI 的内容，请见 `WatchFaceStyle` 类的 API 参考文档。

获得设备屏幕信息

当系统确定了设备屏幕的属性时，系统会调用 `Engine.onPropertiesChanged()` 方法，例如设备是否使用低比特率的环境模式和屏幕是否需要烧毁保护。

下面的代码介绍如何获得这些属性：

```
@Override
public void onPropertiesChanged(Bundle properties) {
    super.onPropertiesChanged(properties);
    mLowBitAmbient = properties.getBoolean(PROPERTY_LOW_BIT_AMBIENT, false);
    mBurnInProtection = properties.getBoolean(PROPERTY_BURN_IN_PROTECTION,
            false);
}
```

当绘制表盘时，我们应该考虑这些设备属性。

- 对于使用低比特率环境模式的设备，屏幕在环境模式下为每种颜色提供更少的比特，所以当设备切换到环境模式时，我们应该禁用抗锯齿和位图滤镜。
- 对于要求烧毁保护的设备，在环境模式下避免使用大块的白色像素，并且不要将内容放在离屏幕边缘 10 个像素范围内，因为系统会周期地改变内容以避免像素烧毁。

更多关于低比特率环境模式和烧毁保护的内容，请见 Optimize for Special Screens。更多关于如何禁用位图滤镜的内容，请见位图滤镜

响应两种模式间的变化

当设备在环境模式和交互模式之间转换时，系统会调用 `Engine.onAmbientModeChanged()` 方法。我们的服务实现应该对在两种模式间切换作出必要的调整，然后调用 `invalidate()` 方法来重新绘制表盘。

下面的代码介绍了这个方法如何在 *WatchFace* 示例的 `AnalogWatchFaceService` 类中实现：

```
@Override
public void onAmbientModeChanged(boolean inAmbientMode) {

    super.onAmbientModeChanged(inAmbientMode);

    if (mLowBitAmbient) {
        boolean antiAlias = !inAmbientMode;
        mHourPaint.setAntiAlias(antiAlias);
        mMinutePaint.setAntiAlias(antiAlias);
        mSecondPaint.setAntiAlias(antiAlias);
        mTickPaint.setAntiAlias(antiAlias);
    }
    invalidate();
    updateTimer();
}
```

这个例子对一些图形风格做出了调整和销毁画布，使得系统可以重新绘制表盘。

绘制表盘

绘制自定义的表盘，系统调用带有 Canvas 实例和绘制表盘所在的 bounds 两个参数的 `Engine.onDraw()` 方法。bounds 参数说明任意内插的区域，如一些圆形设备底部的“下巴”。我们可以像下面介绍的一样来使用画布绘制表盘：

1. 如果是首次调用 `onDraw()` 方法，缩放背景来匹配它。
2. 检查设备处于环境模式还是交互模式。
3. 处理任何图形计算。

4. 在画布上绘制背景位图。
5. 使用 Canvas 类中的方法绘制表盘。

在 *WatchFace* 示例中的 `AnalogWatchFaceService` 类按照如下这些步骤来实现 `onDraw()` 方法：

```
@Override
public void onDraw(Canvas canvas, Rect bounds) {
    // Update the time
    mTime.setToNow();

    int width = bounds.width();
    int height = bounds.height();

    // Draw the background, scaled to fit.
    if (mBackgroundScaledBitmap == null
        || mBackgroundScaledBitmap.getWidth() != width
        || mBackgroundScaledBitmap.getHeight() != height) {
        mBackgroundScaledBitmap = Bitmap.createScaledBitmap(mBackgroundBitmap,
                                          width, height, true /* filter */);
    }
    canvas.drawBitmap(mBackgroundScaledBitmap, 0, 0, null);

    // Find the center. Ignore the window insets so that, on round watches
    // with a "chin", the watch face is centered on the entire screen, not
    // just the usable portion.
    float centerX = width / 2f;
    float centerY = height / 2f;

    // Compute rotations and lengths for the clock hands.
    float secRot = mTime.second / 30f * (float) Math.PI;
    int minutes = mTime.minute;
    float minRot = minutes / 30f * (float) Math.PI;
    float hrRot = ((mTime.hour + (minutes / 60f)) / 6f ) * (float) Math.PI;

    float secLength = centerX - 20;
    float minLength = centerX - 40;
    float hrLength = centerX - 80;

    // Only draw the second hand in interactive mode.
    if (!isInAmbientMode()) {
        float secX = (float) Math.sin(secRot) * secLength;
        float secY = (float) -Math.cos(secRot) * secLength;
        canvas.drawLine(centerX, centerY, centerX + secX, centerY +
                        secY, mSecondPaint);
    }

    // Draw the minute and hour hands.
    float minX = (float) Math.sin(minRot) * minLength;
    float minY = (float) -Math.cos(minRot) * minLength;
    canvas.drawLine(centerX, centerY, centerX + minX, centerY + minY,
                    mMinutePaint);
    float hrX = (float) Math.sin(hrRot) * hrLength;
    float hrY = (float) -Math.cos(hrRot) * hrLength;
    canvas.drawLine(centerX, centerY, centerX + hrX, centerY + hrY,
                    mHourPaint);
}
```

这个方法根据现在的时间计算时钟指针的位置和使用在 `onCreate()` 方法中初始化的图形风格将时钟指针绘制在背景位图之上。其中，秒针只会在交互模式下绘制出来，环境模式不会显示。

更多的关于用 Canvas 实例绘制的内容，请见 Canvas and Drawables。

在 Android SDK 的 *WatchFace* 示例包括附加的表盘，我们可以用作如何实现 `onDraw()` 方法的例子。

在表盘上显示信息

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/information.html

为了显示时间，Android Wear 设备以 cards、notifications 和其它可穿戴应用的形式向用户提供相关的信息。创建自定义表盘不仅可以以丰富的方式显示时间，还可以在用户扫视设备的时候显示相关的信息。

像其它可穿戴应用一样，我们的表盘可以通过可穿戴数据层 API 与可穿戴设备上的应用通信。某些情况下，我们需要在工程中的手持式应用模块里创建一个 activity，该 activity 从互联网或者用户的配置文件中检索数据，然后将数据分享给表盘。



Figure 1. 表盘集成数据的例子

创建丰富的体验

在设计和实现上下文感知的表盘前，先回答下面几个问题：

- 我们想要包含什么类型的数据？
- 我们可以从哪里获得数据？

- 数据多久会显著变化？
- 如何表达数据，使得用户瞥一眼就明白其中的意思？

Android Wear 设备通常与一个带有 GPS 或者蜂窝数据连接的配套设备配对，所以我们有无限的可能来整合不同数据到表盘中，例如位置、日历事件、社交媒体、图片、股票市场报价、新闻事件体育得分等等。然而，并不是所有类型的数据都适合表盘，所以我们需要考虑哪种类型的数据与用户最相关。当可穿戴没有配对的设备或者互联网连接断开时，表盘应该优雅地处理这些情况。

Android Wear 设备上活动的表盘是一个一直在运行的应用，所以我们必须使用高效节能的方法来获取数据。例如，我们每隔10分钟而不是每隔1分钟去获取当前的天气然后将结果保存到本地。当设备从环境模式切换到交互模式时，我们可以刷新上下文数据。这是因为在切换到交互模式时，用户很可能想瞥一眼手表。

由于屏幕的空间有限，并且用户看手表也只是一次看一两秒，所以我们应该在表盘上面将上下文信息归纳起来。有时表达上下文信息最好的方法是用图形和颜色来反应。例如，表盘可以根据当前的天气改变自身的背景图片。

添加数据到表盘

Android SDK 中的 *WatchFace* 示例展示了如何在 `CalendarWatchFaceService` 类里从用户的配置文件中获得日程数据，然后显示接下来的24小时有多少个会议。这个示例位于 `android-sdk/samples/android-21/wearable/WatchFace` 目录下。



Figure 2. 日程表盘

按照下面的步骤实现包含上下文数据的表盘：

1. 提供一个任务来检索数据。
2. 创建一个自定义定时器来周期性地调用任务，或者当外部数据变化时通知表盘服务。
3. 用更新的数据重新绘制表盘。

下面的内容详细介绍了上述几个步骤。

提供一个任务来检索数据

在 `CanvasWatchFaceService.Engine` 实现里创建一个继承 AsyncTask 的类。然后添加用于接收我们感兴趣的数据的代码。

下面是 `CalendarWatchFaceService` 类获取第二天会议数量的代码：

```
/* Asynchronous task to load the meetings from the content provider and
 * report the number of meetings back using onMeetingsLoaded() */
private class LoadMeetingsTask extends AsyncTask<Void, Void, Integer> {
    @Override
    protected Integer doInBackground(Void... voids) {
        long begin = System.currentTimeMillis();
        Uri.Builder builder =
                WearableCalendarContract.Instances.CONTENT_URI.buildUpon();
        ContentUris.appendId(builder, begin);
        ContentUris.appendId(builder, begin + DateUtils.DAY_IN_MILLIS);
        final Cursor cursor = getContentResolver() .query(builder.build(),
                null, null, null, null);
        int numMeetings = cursor.getCount();
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Num meetings: " + numMeetings);
        }
        return numMeetings;
    }

    @Override
    protected void onPostExecute(Integer result) {
        /* get the number of meetings and set the next timer tick */
        onMeetingsLoaded(result);
    }
}
```

Wearable Support 库的 `WearableCalendarContract` 类可以直接存取配套设备用户的日历事件。

当任务检索完数据时，我们的代码会调用一个回调方法。下面的内容详细介绍了如何实现这个回调方法。

更多关于从日历获取数据的内容，请参考 Calendar Provider API 指南。

创建自定义定时器

我们可以实现一个周期计数的自定义定时器来更新数据。`CalendarWatchFaceService` 类使用一个 Handler 实例通过线程的消息队列来发送和处理延时的消息：

```
private class Engine extends CanvasWatchFaceService.Engine {
    ...
    int mNumMeetings;
    private AsyncTask<Void, Void, Integer> mLoadMeetingsTask;

    /* Handler to load the meetings once a minute in interactive mode. */
    final Handler mLoadMeetingsHandler = new Handler() {
        @Override
        public void handleMessage(Message message) {
            switch (message.what) {
                case MSG_LOAD_MEETINGS:
                    cancelLoadMeetingTask();
                    mLoadMeetingsTask = new LoadMeetingsTask();
                    mLoadMeetingsTask.execute();
                    break;
            }
        }
    };
    ...
}
```

当可以看到表盘时，这个方法初始化了定时器：

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);
    if (visible) {
        mLoadMeetingsHandler.sendEmptyMessage(MSG_LOAD_MEETINGS);
    } else {
        mLoadMeetingsHandler.removeMessages(MSG_LOAD_MEETINGS);
        cancelLoadMeetingTask();
    }
}
```

下面的内容介绍在 `onMeetingsLoaded()` 方法设置下一个定时器。

用更新的数据重新绘制表盘

当任务检索完数据时，调用 `invalidate()` 方法使得系统可以重新绘制表盘。将数据保存到 `Engine` 类的成员变量，这样我们就可以在 `onDraw()` 方法中访问数据。

`CalendarWatchFaceService` 类提供一个回调方法给任务在检索完日程数据后调用：

```
private void onMeetingsLoaded(Integer result) {
    if (result != null) {
        mNumMeetings = result;
        invalidate();
    }
    if (isVisible()) {
        mLoadMeetingsHandler.sendEmptyMessageDelayed(
                MSG_LOAD_MEETINGS, LOAD_MEETINGS_DELAY_MS);
    }
}
```

回调方法将结果保存在一个成员变量中，销毁 view，和调度下一个定时器再次运行任务。

提供配置 Activity

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/configuration.html

当用户安装一个包含表盘的可穿戴应用的手持式应用时，它们可以在手持式设备上的 Android Wear 配套应用和在可穿戴设备上的表盘选择器中使用。用户可以在配套应用上或者在可穿戴设备的表盘选择器上选择使用哪个表盘。

一些表盘提供配置参数，让用户客制化表盘的外观和行为。例如，一些表盘让用户选择自定义的背景颜色，另一些表盘提供两个不同时区的时间，使得用户可以选择感兴趣的时区。

提供配置参数的表盘让用户通过可穿戴应用的一个 activity、手持应用的一个 activity或者两者的 activity 来客制化表盘。用户可以启动可穿戴设备上的可穿戴配置 activity，他们也可以启动 Android Wear 配套应用的配套配置 activity。

Android SDK 中 *WatchFace* 示例的数字表盘介绍了如何实现手持式和可穿戴配置 activity 和如何应配置变化而更新表盘。这个示例位于 `android-sdk/samples/android-21/wearable/WatchFace` 目录。

指定配置 activity 的 Intent

如果表盘包括配置的 activity，那么添加下面的元数据项到可穿戴应用 manifest 文件的服务声明部分：

```
<service
    android:name=".DigitalWatchFaceService" ... />
    <!-- companion configuration activity -->
    <meta-data
        android:name=
           "com.google.android.wearable.watchface.companionConfigurationAction"
        android:value=
           "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
    <!-- wearable configuration activity -->
    <meta-data
        android:name=
           "com.google.android.wearable.watchface.wearableConfigurationAction"
        android:value=
           "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
    ...
</service>
```

在应用的包名之前定义这些元数据项的值。配置 activity 为这个 intent 注册 intent filters，然后系统在用户想配置表盘时启动这个 intent。

如果表盘只包括一个配套或者可穿戴配置 activity，那么我们只需要包括上述例子响应的元数据项。

创建可穿戴配置 activity

可穿戴配置 activity 提供了有限组表盘客制化选择，这是因为复杂的菜单在小屏幕上很难导航。我们的可穿戴配置 activity 应该提供二元选择和很少的选项来客制化表盘主要的方面。

为了创建一个可穿戴配置 activity，添加一个新的 activity 到可穿戴应用并且在可穿戴应用的 manifest 文件中声明下面的 intent filter：

```
<activity
    android:name=".DigitalWatchFaceWearableConfigActivity"
    android:label="@string/digital_config_name">
    <intent-filter>
        <action android:name=
            "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
        <category android:name=
        "com.google.android.wearable.watchface.category.WEARABLE_CONFIGURATION" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

这个 intent filter 的 action 的名字必须与之前在指定配置 activity 的 Intent定义的 intent 名字一样。

在我们的配置 activity 中，构建一个简单的 UI 为用户提供选择来客制化表盘。当用户做出选择时，使用可穿戴数据层 API传达配置的变化给表盘 activity。

更多详细内容，请见 *WatchFace* 示例中的 `DigitalWatchFaceWearableConfigActivity` 和 `DigitalWatchFaceUtil` 类。

创建配套配置 activity

配套配置 activity 让用户可以访问全套表盘客制化选择，这是因为在手持式设备更大的屏幕上，用户更加容易与复杂的菜单互动。例如，手持设备上的一个配置 activity 向用户显示复杂的颜色选择器，让用户从该选择器中选择表盘的背景颜色。

为了创建配套配置 activity，添加一个新的 activity 到手持应用并且在手持应用的 manifest 文件中声明下面的 intent filter：

```
<activity
    android:name=".DigitalWatchFaceCompanionConfigActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name=
            "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
        <category android:name=
        "com.google.android.wearable.watchface.category.COMPANION_CONFIGURATION" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

在我们的配置 activity 中，构建一个 UI 为用户提供选项来客制化表盘所有的可配置组件。当用户做出选择时，使用可穿戴数据层 API传达配置的变化给表盘 activity。

更多详细内容，请见 *WatchFace* 示例中的 `DigitalWatchFaceCompanionConfigActivity` 类。

在可穿戴应用中创建一个监听器服务

为了接收配置 activity 中已更新的配置参数，需要在可穿戴应用创建一个服务来实现可穿戴数据层 API 的 `WearableListenerService` 接口。我们的表盘实现可以在配置参数改变时重新绘制表盘。

更多详细内容，请见 *WatchFace* 示例的 `DigitalWatchFaceConfigListenerService` 和 `DigitalWatchFaceService` 类。

定位常见的问题

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/issues.html

创建 Android Wear 的客制化表盘与创建 notification 和可穿戴特有的 activity 的方法不同。这几课介绍如何解决我们在实现第一个表盘时会遇到的一些问题。

检测屏幕的形状

一些 Android Wear 设备的屏幕是方形的，另一些是圆形的。圆形屏幕的设备可以在屏幕的底部包含一个插入部分（或者“下巴”）。我们的表盘应该适应和利用好屏幕特定的形状，如 设计指南 中的描述。

Android Wear 让表盘在运行时决定屏幕的形状。为了检测屏幕是方形还是圆形，需要像下面的代码一样重写 `CanvasWatchFaceService.Engine` 类的 `onApplyWindowInsets()` 方法：

```
private class Engine extends CanvasWatchFaceService.Engine {
    boolean mIsRound;
    int mChinSize;

    @Override
    public void onApplyWindowInsets(WindowInsets insets) {
        super.onApplyWindowInsets(insets);
        mIsRound = insets.isRound();
        mChinSize = insets.getSystemWindowInsetBottom();
    }
    ...
}
```

当重新绘制表盘时，检查成员变量 `mIsRound` 和 `mChinSize` 的值来适应我们的设计。

容纳 Card

当用户接收到一个 notification，notification card 可能会遮盖屏幕很大一部分，这取决于系统 UI 的风格。表盘应该适应这些情况，确保当 notification card 出现时用户仍然可以看到时间。

当 notification card 出现时，模拟表盘需要调整，如缩小表盘使得自身不被 card 覆盖。数字表盘在屏幕显示时间的区域不会被 card 覆盖，通常不需要作出调整。使用 `WatchFaceService.getPeekCardPosition()` 方法确定在 card 上方可用于调整表盘的空间。



Figure 1. 当 notification card 出现时，一些模拟表盘需要调整

在环境模式下，card 的背景是透明的。如果我们的表盘在环境模式下，card 的附近包含详细的信息，那么可以考虑在 card 的上面绘制一个黑色方块，确保用户可以读到 card 的内容。

配置系统指示图标

为了确保系统指示图标一直可见，当创建一个 `WatchFaceStyle` 实例时，我们可以将配置系统指示图标在屏幕的位置和决定是否需要背景保护：

- 使用 `setStatusBarGravity()` 方法设置状态栏的位置。
- 使用 `setHotwordIndicatorGravity()` 方法设置热词的位置。
- 使用 `setViewProtection()` 方法，用一个灰色的半透明背景保护状态栏和热词。由于系统指示图标是白色的，如果我们的表盘背景是明亮的，这样做事很必要的。



Figure 2. 状态栏

更多关于系统指示图标的内容，请查看配置系统 UI 和 设计指南。

使用相对尺寸

不同厂商的 Android Wear 设备屏幕会有不同的尺寸和分辨率。我们的表盘应该通过使用相对尺寸而不是绝对像素值来适应这些差异。

当我们绘制表盘时，用 Canvas.getWidth()) 和 Canvas.getHeight()) 方法获得画布的尺寸，然后以屏幕尺寸一部分所占比例的值来设置图片的位置。如果重新绘制表盘的组件来响应 card，那么根据屏幕里 card 上方剩下空间所占比例的值来重新绘制表盘。

优化性能和电池使用时间

> 编写:heray1990 - 原文: http://developer.android.com/training/wearables/watch-faces/performance.html

除了有好的 notification cards 和系统指示图标之外，我们还需要确保表盘的动画运行流畅，服务不会执行没必要的计算。Android Wear 的表盘会在设备上一直运行，所以表盘高效地使用电池显得十分重要。

这节课提供了一些提示来加快动画的速度，测量和节省设备上的电量。

减小位图资源的尺寸

很多表盘由一张背景图片及其它被转换和覆盖在背景图片上面的图形资源组成，例如时钟指针和其它随着时间移动的设计组件。没词系统重新绘制表盘的时候，在 `Engine.onDraw()` 方法里面，这些图像组件往往会旋转（有时会缩放），详见绘制表盘。

这些图形资源越大，转换它们所需的运算量就越大。在 `Engine.onDraw()` 方法中转换大的图形资源会大大地减低系统运行动画的帧率。

为了提升表盘的性能，我们需要：

- 不要使用比我们需要的更大的图像组件。
- 删除边缘周围多出来的透明像素。

在 Figure 1 中的时钟指针例子可以将大小减小97%。

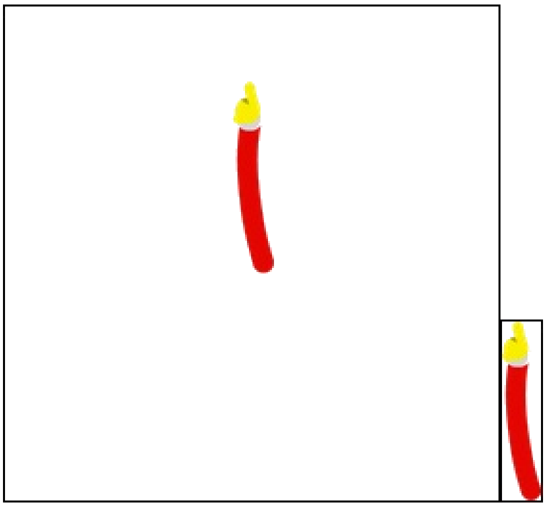


Figure 1. 可以剪裁多余像素的时钟指针

这节内容介绍的减小位图资源的大小不仅提升了动画的性能，也节省了电量。

合并位图资源

如果我们有经常需要一起绘制的位图，那么可以考虑将它们合并到同一个图形资源中。在交互模式下，通常我们可以将背景图片和计数标记组合起来，从而避免没词系统重新绘制表盘时，都去绘制两个全屏的位图。

当绘制可缩放的位图时禁用反锯齿功能

当使用 Canvas.drawBitmap() 方法绘制可缩放的位图，我们可以使用 Paint 实例去设置一些选项。为了提升性能，使用 setAntiAlias()) 方法禁用反锯齿，这是由于这个设置对于位图没有任何影响。

使用位图滤镜

对于绘制在其它组件上的位图资源，可以在同一个 Paint 实例上使用 setFilterBitmap()) 方法来打开位图滤镜。Figure 2显示了使用和没使用位图滤镜的放大的时钟指针。

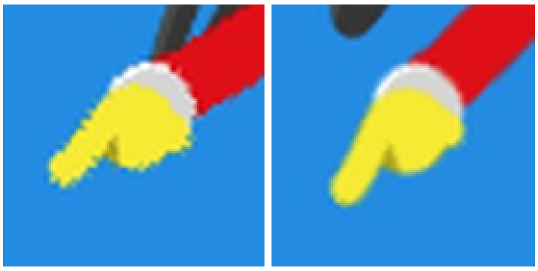


Figure 2. 没使用位图滤镜（左）和使用位图滤镜（右）

> **Note:** 在低比特率的环境模式下，系统不能可靠地渲染图片的颜色，从而不能保证成功地执行位图滤镜。因此，在环境模式下，禁用位图滤镜。

将复杂的操作移到 onDraw() 方法外面

每次重新绘制表盘时，系统会调用 `Engine.onDraw()` 方法，所以为了提升性能，我们应该只将用于更新表盘的重要的操作放到这个方法中。

可以的话，避免在 `onDraw()` 方法里处理下面这些操作：

- 加载图片和其它资源。
- 调整图片的大小。
- 分配对象。
- 运行在帧与帧之间不会改变的计算。

通常可以在 `Engine.onCreate()` 方法中运行上述这些操作。我们可以在 执行Engine.onSurfaceChanged() 方法之前调整图片大小。其中，该方法提供了画布的大小。

为了分析表盘的性能，我们可以使用 Android Device Monitor。特别地，确保 `Engine.onDraw()` 实现的运行时间是短的和调用是一致的。详细内容见使用 DDMS。

节能的最佳做法

除了前面部分介绍的技术之外，我们还需要按照下面的最佳做法来降低表盘的电量消耗。

降低动画的帧频

动画通常需要消耗大量计算资源和电量。大部分动画在每秒30帧的情况下看上去是流畅的，所以我们应该避免动画的帧频比每秒30帧高。

让 CPU 睡眠

表盘的动画和内容的小变化会唤醒 CPU。表盘应该在动画之间让 CPU 睡眠。例如，在交互模式下，我们可以每隔一秒使用动画的短脉冲，然后在下一秒让 CPU 睡眠。频繁地让 CPU 睡眠，甚至短暂地，都可以有效地降低电量消耗。

为了最大化电池使用时间，谨慎地使用动画。即使闪烁动画在闪烁的时候也会唤醒 CPU 并且消耗电量。

监控电量消耗

在 Android Wear companion app 的 **Settings > Watch battery** 下，开发者和用户可以看到可穿戴设备中不同进程还有多少电量。

在 Android 5.0 中，更多关于提升电池使用时间的信息，请见 Project Volta。

Android Wear 上的位置检测

> 编写:heray1990 - 原文: http://developer.android.com/training/articles/wear-location-detection.html

可穿戴设备上的位置感知让我们可以创建为用户提供更好地了解地理位置、移动和周围事物的应用。由于可穿戴设备小型和方便的特点，我们可以构建低摩擦应用来记录和响应位置数据。

一些可穿戴设备带有 GPS 感应器，它们可以在不需要其它设备的帮助下检索位置数据。无论如何，当我们在可穿戴应用上请求获取位置数据，我们不需要担心位置数据从哪里发出；系统会用最节能的方法来检索位置更新。我们的应用应该可以处理位置数据的丢失，以防没有内置 GPS 感应器的可穿戴设备与配套设备断开连接。

这篇文章介绍如何检查设备上的位置感应器、检索位置数据和监视数据连接。

> **Note:** 这篇文章假设我们知道如何使用 Google Play services API 来检索位置数据。更多相关的内容，请见 Android 位置信息。

连接 Google Play Services

可穿戴设备上的位置数据可以通过 Google Play services location APIs 来获取。我们可以使用 FusedLocationProviderApi 和它伴随的类来获取这个数据。为了访问位置服务，可以创建 GoogleApiClient 实例，这个实例是任何 Google Play services APIs 的主要入口。

> **Caution:** 不要使用 Android 框架已有的 Location APIs。检索位置更新最好的方法是通过这篇文章介绍的 Google Play services API 获取。

为了连接 Google Play services，配置应用来创建 GoogleApiClient 实例：

1. 创建一个 activity 来指定 ConnectionCallbacks、OnConnectionFailedListener 和 LocationListener 接口的实现。
2. 在 activity 的 onCreate()) 方法中，创建 GoogleApiClient 实例和添加位置服务。
3. 为了优雅地管理连接的生命周期，在 onResume()) 方法里调用 connect()) 和在 onPause()) 方法里调用 disconnect())。

下面的代码示例介绍了一个 activity 的实现来实现 LocationListener 接口：

```
public class WearableMainActivity extends Activity implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener,
    LocationListener {

    private GoogleApiClient mGoogleApiClient;
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ...
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(LocationServices.API)
                .addApi(Wearable.API)  // used for data layer API
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onResume() {
        super.onResume();
        mGoogleApiClient.connect();
        ...
    }

    @Override
    protected void onPause() {
        super.onPause();
        ...
        mGoogleApiClient.disconnect();
    }
}
```

更多关于连接 Google Play services 的内容，请见 Accessing Google APIs。

请求位置更新

应用连接到 Google Play services API 之后，它已经准备好开始接收位置更新了。当系统为我们的客户端调用 onConnected() 回调函数时，我们可以按照下面的步骤构建位置更新请求：

1. 创建一个 LocationRequest 对象并且用像 setPriority() 这样的方法设置选项。
2. 使用 requestLocationUpdates() 请求位置更新。
3. 在 onPause() 方法里使用 removeLocationUpdates() 删除位置更新。

下面的例子介绍了如何接收和删除位置更新：

```
@Override
public void onConnected(Bundle bundle) {
    LocationRequest locationRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
            .setInterval(UPDATE_INTERVAL_MS)
            .setFastestInterval(FASTEST_INTERVAL_MS);

    LocationServices.FusedLocationApi
            .requestLocationUpdates(mGoogleApiClient, locationRequest, this)
            .setResultCallback(new ResultCallback() {

                @Override
                public void onResult(Status status) {
                    if (status.getStatus().isSuccess()) {
                        if (Log.isLoggable(TAG, Log.DEBUG)) {
                            Log.d(TAG, "Successfully requested location updates");
                        }
                    } else {
                        Log.e(TAG,
                                "Failed in requesting location updates, "
                                        + "status code: "
                                        + status.getStatusCode()
                                        + ", message: "
                                        + status.getStatusMessage());
                    }
                }
            });
}

@Override
protected void onPause() {
    super.onPause();
    if (mGoogleApiClient.isConnected()) {
        LocationServices.FusedLocationApi
             .removeLocationUpdates(mGoogleApiClient, this);
    }
    mGoogleApiClient.disconnect();
}

@Override
public void onConnectionSuspended(int i) {
    if (Log.isLoggable(TAG, Log.DEBUG)) {
        Log.d(TAG, "connection to location client suspended");
    }
}
```

至此，我们已经打开了位置更新，系统调用 onLocationChanged()) 方法，同时按照 setInterval()) 指定的时间间隔更新位置。

检测设备上的 GPS

不是所有的可穿戴设备都有 GPS 感应器。如果用户出去外面并且将他们的手机放在家里，那么我们的可穿戴应用无法通过一个绑定连接来接收位置数据。如果可穿戴设备没有 GPS 感应器，那么我们应该检测到这种情况并且警告用户位置功能不可用。

使用 hasSystemFeature()) 方法确定 Android Wear 设备是否有内置的 GPS 感应器。下面的代码用于当我们启动一个 activity 时，检测设备是否有内置的 GPS 感应器：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main_activity);
    if (!hasGps()) {
        Log.d(TAG, "This hardware doesn't have GPS.");
        // Fall back to functionality that does not use location or
        // warn the user that location function is not available.
    }

    ...
}

private boolean hasGps() {
    return getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS);
}
```

处理断开事件

可穿戴设备在回答绑定连接位置数据时可能会突然断开连接。如果我们的可穿戴应用期待持续的数据，那么我们必须处理数据中断或者不可用的断线问题。在一个不带有 GPS 感应器的可穿戴设备上，当设备与绑定数据连接断开时，位置数据会丢失。

以防基于绑定位置数据连接的应用和可穿戴设备没有 GPS 感应器，我们应该检测连接的断线，警告用户和优雅地降低应用的功能。

为了检测数据连接的断线：

1. 继承 WearableListenerService 来监听重要的数据层事件。
2. 在 Android manifest 文件中声明一个 intent filter 来把WearableListenerService 通知给系统。这个 filter 允许系统按需绑定我们的服务。

```
<service android:name=".NodeListenerService">
 <intent-filter>
     <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
 </intent-filter>
</service>
```

3. 实现 onPeerDisconnected()) 方法并处理设备是否有内置 GPS 的情况。

```
public class NodeListenerService extends WearableListenerService {

    private static final String TAG = "NodeListenerService";

    @Override
    public void onPeerDisconnected(Node peer) {
        Log.d(TAG, "You have been disconnected.");
        if(!hasGPS()) {
            // Notify user to bring tethered handset
            // Fall back to functionality that does not use location
        }
    }
    ...
}
```

更多相关的信息，请见 监听数据层事件 指南。

处理找不到位置的情况

当 GPS 信号丢失了，我们仍然可以使用 getLastLocation()) 检索最后可知位置。这个方法在我们无法修复 GPS 连接或者设备没有内置 GPS 并且断开与手机连接的情况下很有用。

下面的代码使用 getLastLocation()) 检索最后可知位置：

```
Location location = LocationServices.FusedLocationApi
                .getLastLocation(mGoogleApiClient);
```

同步数据

如果可穿戴应用使用内置 GPS 记录数据，那么我们可能想要与手持应用同步位置数据。对于 LocationListener，我们可以实现 onLocationChanged()) 方法来检测和记录它改变的位置。

下面的可穿戴应用代码检测位置变化和使用数据层 API 来保存用于手机应用日后检索的数据：

```
@Override
public void onLocationChanged(Location location) {
    ...
    addLocationEntry(location.getLatitude(), location.getLongitude());

}

private void addLocationEntry(double latitude, double longitude) {
    if (!mSaveGpsLocation || !mGoogleApiClient.isConnected()) {
        return;
    }

    mCalendar.setTimeInMillis(System.currentTimeMillis());

    // Set the path of the data map
    String path = Constants.PATH + "/" + mCalendar.getTimeInMillis();
    PutDataMapRequest putDataMapRequest = PutDataMapRequest.create(path);

    // Set the location values in the data map
    putDataMapRequest.getDataMap()
            .putDouble(Constants.KEY_LATITUDE, latitude);
    putDataMapRequest.getDataMap()
            .putDouble(Constants.KEY_LONGITUDE, longitude);
    putDataMapRequest.getDataMap()
            .putLong(Constants.KEY_TIME, mCalendar.getTimeInMillis());

    // Prepare the data map for the request
    PutDataRequest request = putDataMapRequest.asPutDataRequest();

    // Request the system to create the data item
    Wearable.DataApi.putDataItem(mGoogleApiClient, request)
            .setResultCallback(new ResultCallback() {
                @Override
                public void onResult(DataApi.DataItemResult dataItemResult) {
                    if (!dataItemResult.getStatus().isSuccess()) {
                        Log.e(TAG, "Failed to set the data, "
                                + "status: " + dataItemResult.getStatus()
                                .getStatusCode());
                    }
                }
            });
}
```

更多关于如何使用数据层 API 的内容，请见 发送与同步数据 指南。

创建TV应用

> 编写:applepyarc - 原文:http://developer.android.com/training/tv/index.html

以下课程将教授如何为TV设备开发应用。

> **Note**：如何在Google Play发布你的TV应用，详细请参考Distributing to Android TV。

创建TV应用

如何开发TV应用和移植已有应用到TV设备。

创建TV播放应用

如何开发提供媒体目录和播放内容的应用。

帮助用户在TV上找到内容

如何帮助用户从你的应用发现所需内容。

创建TV游戏应用

如何开发TV游戏。

创建TV直播应用

如何开发TV直播应用。

TV应用清单

TV应用的需求清单

创建TV应用

> 编写:applepyarc - 原文:http://developer.android.com/training/tv/start/index.html

- Android 5.0(API 21)或以上

Android提供丰富的用户体验，优化应用运行于诸如高清电视等大屏幕设备。TV应用有机会为沙发上的用户提供愉快的体验。

TV应用使用与手机或平板应用相同的架构。这意味着你可以基于已知的Android应用开发来创建新的TV应用。或者移植已有的应用到TV设备上。但是，在UI上，TV和手机或平板大不相同。为了使应用顺畅地运行在TV设备上，我们必须设计能够在即使3米之外也易于理解的新界面，提供可以使用方向键和选择键操作的导航界面。

以下课程描述了如何开始创建TV应用，包括设置开发环境，界面及导航的基本要求，以及如何处理TV设备通常不具备的硬件特性。

> **Note**: 鼓励使用Android Studio创建TV应用，因为它提供了创建项目的步骤，库包含和快捷打包。本课程假设你正在使用Android Studio。

课程

- **创建TV应用的第一步**

  学习如何为TV应用创建一个新的Android Studio项目或者修改已有的应用运行到TV设备上。

- **处理TV硬件**

  学习如何检查应用是否运行在TV硬件上，处理不支持的硬件特性和管理控制器设备。

- **创建TV布局**

  学习TV界面的最小要求及其实现。

- **创建TV导航**

  学习TV导航的最小要求以及如何实现TV兼容的导航。

创建TV应用的第一步 >

创建TV应用的第一步

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/start/start.html

TV应用使用与手机和平板同样的架构。这种相似性意味着我们可以修改现有的应用到TV设备或者用以前安卓应用的经验开发TV应用。

> **Important**: 想把Android TV应用放在Google Play中应满足一些特定要求。更多信息, 参考TV App Quality中的要求列表。

本课程介绍如何准备TV应用开发环境,和使应用能够运行在TV设备上的最低要求。

查明支持的媒体格式

查看以下文档信息，包括代码，协议和Android TV支持的格式。

- 支持的媒体格式
- DRM
- android.drm
- ExoPlayer
- android.media.MediaPlay

查明支持的媒体格式

查看一下文档关于代码，协议和Android TV支持的格式。

- 支持的媒体格式
- DRM
- android.drm
- ExoPlayer
- android.media.MediaPlay

创建TV项目

本节讨论如何修改已有的应用或者新建一个应用使之能够运行在电视设备上。在TV设备上运行的应用必须使用这些主要组件:

- **Activity for TV** (必须) - 在您的application manifest中, 声明一个可在TV设备上运行的activity。

- **TV Support Libraries** (可选) - 这些支持库Support Libraries 可以提供搭建TV用户界面的控件。

前提条件

在创建TV应用前，必须做以下事情:

- 更新SDK tools到版本24.0.0或更高 更新的SDK工具能确保编译和测试TV应用
- 更新SDK为Android 5.0 (API 21)或更高 更新的平台版本为TV应用提供更新的API
- 创建或更新应用工程 为了支持TV新API，我们必须创建一个新工程或者修改原工程的目标平台为Android 5.0 (API版本21)或者更高。

声明一个TV Activity

一个应用想要运行在TV设备中，必须在它的manifest中定义一个启动activity，用intent filter包含CATEGORY_LEANBACK_LAUNCHER。这个filter表明你的应用是在TV上可用，并且为Google Play上发布TV应用所必须。定义这个intent也意味着点击主屏幕的应用图标时，就是打开的这个activity。

接下来的代码片段显示如何在manifest中包含这个intent filter：

```
<application
  android:banner="@drawable/banner" >
  ...
  <activity
    android:name="com.example.android.MainActivity"
    android:label="@string/app_name" >

    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>

  <activity
    android:name="com.example.android.TvActivity"
    android:label="@string/app_name"
    android:theme="@style/Theme.Leanback">

    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LEANBACK_LAUNCHER" />
    </intent-filter>

  </activity>
</application>
```

例子中第二个activity manifest定义的activity是TV设备中的一个启动入口。

> **Caution**：如果在你的应用中不包含CATEGORY_LEANBACK_LAUNCHER intent filter，它不会出现在TV设备的Google Play商店中。并且，即使你把不包含此filter的应用用开发工具装载到TV设备中，应用仍然不会出现在TV用户界面上。

如果你正在为TV设备修改现有的应用，就不应该与手机和平板用同样的activity布局。TV的用户界面（或者现有应用的TV部分）应该提供一个更简单的界面，更容易坐在沙发上用遥控器操作。TV应用设计指南，参考TV Design指导。查看TV界面布局的最低要求，参考：Building TV Layouts。

声明Leanback支持

Android TV需要你的应用使用Leanback用户界面。如果你正在开发一个运行在移动设备（手机，可穿戴，平板等等）也包括TV的应用，设置 `required` 属性为 `false` 。因为如果设置为 `true` ，你的应用将仅能运行在用Leanback UI的设备上。

```
<manifest>
    <uses-feature android:name="android.software.leanback"
        android:required="false" />
    ...
</manifest>
```

声明不需要触屏

运行在TV设备上的应用不依靠触屏去输入。为了清楚表明这一点，TV应用的manifest必须声明 `android.hardware.touchscreen` 为不需要。这个设置表明应用能够工作在TV设备上，并且也是Google Play认定你的应用为TV应用的要求。接下来的示例代码展示这个manifest声明：

```
<manifest>
    <uses-feature android:name="android.hardware.touchscreen"
              android:required="false" />
    ...
</manifest>
```

> **Caution**：必须在manifest中声明触屏是不需要的，否则应用不会出现在TV设备的Google Play商店中。

提供一个主屏幕横幅

如果应用包含一个Leanback的intent filter，它必须提供每个语言的主屏幕横幅。横幅是出现在应用和游戏栏的主屏的启动点。在manifest中这样描述横幅：

```
<application
    ...
    android:banner="@drawable/banner" >

    ...
</application>
```

在 `application` 中添加 `android:banner` 属性为所有的应用activity提供默认的横幅，或者在特定activity的 `activity` 中添加横幅。

在UI模式和TV设计指导中查看Banners。

添加TV支持库

Android SDK包含用于TV应用的支持库。这些库为TV设备提供API和用户界面控件。这些库位于 `<sdk>/extras/android/support/` 目录。以下是这些库的列表和它们的作用介绍：

- v17 leanback library - 提供TV应用的用户界面控件，特别是用于媒体播放应用的控件。
- v7 recyclerview library - 提供了内存高效方式的长列表的管理显示类。有一些v17 leanback库的类依赖于本库的类。
- v7 cardview library - 提供显示信息卡的用户界面控件，如媒体图片和描述。

> **Note**：TV应用中可以不用这些库。但是，我们强烈推荐使用它们，特别是为应用提供媒体目录浏览界面时。

如果我们决定用 `v17 leanback library` ，我们应该注意它依赖于v4 support library。这意味着要用leanback支持库必须包含以下所有的支持库：

- v4 support library
- v7 recyclerview support library
- v17 leanback support library

`v17 leanback library` 包含资源文件，需要你在应用中采取特定的步骤去包含它。插入带资源文件的支持库的说明，查看Support Library Setup。

创建TV应用

在完成上面的步骤之后，到了给大屏幕创建应用的时候了！检查一下这些额外的专题可以帮助我们创建TV应用：

- 创建TV播放应用 - TV就是用来娱乐的，因此安卓提供了一套用户界面工具和控件，用来创建视频和音乐的TV应用，并且让用户浏览想看到的内容。
- 帮助用户找到TV内容 - 因为所有的内容选择都用手指操作遥控器，所以帮助用户找到想

要的内容几乎和提供内容同样重要。这个主题讨论如何在TV设备中处理内容。
- TV游戏 - TV设备是非常好的游戏平台。参考这个主题去创造更好的TV游戏体验。

运行TV应用

运行应用是在开发过程中的一个重要的部分。在安卓SDK中的AVD管理器提供了创建虚拟TV设备的功能，可以让应用在虚拟设备中运行和测试。

创建一个虚拟TV设备

1. 打开AVD管理器。更多信息，参考AVD管理器帮助。
2. 在AVD管理器窗口，点击**Device Definitions**标签。
3. 选择一个Android TV设备描述，并且点击**Create AVD**。
4. 选择模拟器选项并且点击**OK**创建AVD。

> **Note**：获得TV模拟器设备的最佳性能，打开**Use Host GPU option**，支持虚拟设备加速。更多模拟器硬件加速信息，参考Using the Emulator。

在虚拟设备中测试应用

1. 在开发环境中编译TV应用。
2. 从开发环境中运行应用并选择目标为TV虚拟设备。

更多模拟器信息：Using the Emulator。 用Android Studio部署应用到模拟器，查看Debugging with Android Studio。用带ADT插件的Eclipse部署应用到模拟器，查看Building and Running from Eclipse with ADT 。

下一节: 处理TV硬件 >

处理TV硬件

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/start/hardware.html

TV硬件和其他Android设备有实质性的不同。TV不包含一些其他Android设备具备的硬件特性，如触摸屏，摄像头，和GPS。TV操作也完全依赖于其他辅助硬件设备。为了让用户与TV应用交互，他们必须使用遥控器或者游戏手柄。当我们创建TV应用时，必须小心的考虑到TV硬件的限制和操作要求。

本节课程讨论如何检查应用是不是运行在TV上，怎样去处理不支持的硬件特性，和讨论处理TV设备控制器的要求。

TV设备的检测

如果我们创建的应用同时支持TV设备和其他设备，我们可能需要检测应用当前运行在哪种设备上，并调整应用的执行。例如，如果有一个应用通过Intent启动，应用应该检查设备特性然后决定是应该启动TV方面的activity还是手机的activity。

检查应用是否运行在TV设备上，推荐的方式是用UiModeManager.getCurrentModeType())方法检测设备是否运行在TV模式。下面的示例代码展示了如何检查应用是否运行在TV设备上：

```
public static final String TAG = "DeviceTypeRuntimeCheck";

UiModeManager uiModeManager = (UiModeManager) getSystemService(UI_MODE_SERVICE);
if (uiModeManager.getCurrentModeType() == Configuration.UI_MODE_TYPE_TELEVISION) {
    Log.d(TAG, "Running on a TV Device")
} else {
    Log.d(TAG, "Running on a non-TV Device")
}
```

处理不支持的硬件特性

基于应用的设计和功能，我们可能需要在某些硬件特性不可用的情况下工作。这节讨论哪些硬件特性对于TV是典型不可用的，如何去检测缺少的硬件特性，并且去用这些特性的推荐替代方法。

不支持的TV硬件特性

TV和其他设备有不同的目的，因此它们没有一些其他Android设备通常有的硬件特性。由于这个原因，TV设备的Android系统不支持以下特性：

硬件	**Android**特性描述
触屏	`android.hardware.touchscreen`
触屏模拟器	`android.hardware.faketouch`
电话	`android.hardware.telephony`
摄像头	`android.hardware.camera`
蓝牙	`android.hardware.bluetooth`
近场通讯（NFC）	`android.hardware.nfc`
GPS	`android.hardware.location.gps`
麦克风 **[1]**	`android.hardware.microphone`
传感器	`android.hardware.sensor`

> **[1]** 一些TV控制器有麦克风，但不是这里描述的麦克风硬件特性。控制器麦克风是完全被支持的。

查看Features Reference获得完全的特性和子特性列表，和它们的描述。

声明TV硬件需求

Android应用能通过在manifest中定义硬件特性需求来确保应用不能被安装在不提供这些特性的设备上。如果我们正在扩展应用到TV上，仔细地审查我们的manifest的硬件特性需求，它有可能阻止应用安装到TV设备上。

即使我们的应用使用了TV上不存在的硬件特性（如触屏或者摄像头），应用也可以在没有那些特性的情况下工作，需要修改应用的manifest来表明这些特性不是必须的。接下来的manifest代码片段示范了如何声明在TV设备中不可用的硬件特性，尽管我们的应用在非TV设备上可能会用上这些特性。

```
<uses-feature android:name="android.hardware.touchscreen"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.faketouch"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.telephony"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.camera"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.bluetooth"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.nfc"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.gps"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.microphone"
        android:required="false"></uses>
<uses-feature android:name="android.hardware.sensor"
        android:required="false"></uses>
```

> **Note**：一些特性有子特性，如 `android.hardware.camera.front` ，参考：Feature Reference。确保应用中任何子特性也标记为 `required="false"` 。

所有想用在TV设备上的应用必须声明触屏特性不被需要，在创建TV应用的第一步有描述。如果我们的应用使用了一个或更多的上面列表上的特性，改变manifest特性的 `android:required` 属性为 `false` 。

> **Caution**：表明一个硬件特性是必须的，设置它的值为 `true` 可以阻止应用在TV设备上安装或者出现在AndroidTV的主屏幕启动列表上。

一旦我们决定了应用的硬件特性选项，那就必须检查在运行时这些特性的可用性，然后调整应用的行为。下一节讨论如何检查硬件特性和改变应用行为的建议处理。

更多关于filter和在manifest里声明特性，参考：uses-feature。

声明权限会隐含硬件特性

一些uses-permission manifest声明隐含了硬件特性。这些行为意味着在应用中请求一些权限能导致应用不能安装和使用在TV设备上。下面普通的权限请求包含了一个隐式的硬件特性需求：

权限	隐式的硬件需求
RECORD_AUDIO	`android.hardware.microphone`
CAMERA	`android.hardware.camera` *and* `android.hardware.camera.autofocus`
ACCESS_COARSE_LOCATION	`android.hardware.location` *and* `android.hardware.location.network`
ACCESS_FINE_LOCATION	`android.hardware.location` *and* `android.hardware.location.gps`

包含隐式硬件特性需求的完整权限需求列表，参考：uses-feature。如果我们的应用请求了上面列表上的特性的任何一个，在manifest中设置它的隐式硬件特性为不需要（`android:required="false"`）。

检查硬件特性

在应用运行时，Android framework能告诉硬件特性是否可用。用hasSystemFeature(String))方法在运行时检查特定的特性。这个方法只需要一个字符串参数，即想检查的特性名字。

接下来的示例代码展示了如何在运行时检测硬件特性的可用性：

```
// Check if the telephony hardware feature is available.
if (getPackageManager().hasSystemFeature("android.hardware.telephony")) {
    Log.d("HardwareFeatureTest", "Device can make phone calls");
}

// Check if android.hardware.touchscreen feature is available.
if (getPackageManager().hasSystemFeature("android.hardware.touchscreen")) {
    Log.d("HardwareFeatureTest", "Device has a touch screen.");
}
```

触屏

因为大部分的TV没有触摸屏，在TV设备上，Android不支持触屏交互。此外，用触屏交互和坐在离显示器3米外观看是相互矛盾的。

在TV设备中，我们应该设计出支持遥控器方向键（D-pad）远程操作的交互模式。更多关于正确地支持TV友好的控制器操作的信息，参考Creating TV Navigation。

摄像头

尽管TV通常没有摄像头，但是我们仍然可以提供拍照相关的TV应用，如果应用有拍照，查看和编辑图片功能，在TV上可以关闭拍照功能但仍可以允许用户查看甚至编辑图片。如果我们决定在TV上使用摄像相关的应用，在manifest里添加接下来的特性声明：

```
<uses-feature android:name="android.hardware.camera" android:required="false" ></uses>
```

如果在缺少摄像头情况下运行应用，在我们应用中添加代码去检测是否摄像头特性可用，并且调整应用的操作。接下来的示例代码展示了如何检测一个摄像头的存在：

```
// Check if the camera hardware feature is available.
if (getPackageManager().hasSystemFeature("android.hardware.camera")) {
    Log.d("Camera test", "Camera available!");
} else {
    Log.d("Camera test", "No camera available. View and edit features only.");
}
```

GPS

TV是固定的室内设备，并且没有内置的全球定位系统（GPS）接收器。如果我们应用使用定位信息，我们仍可以允许用户搜索位置，或者用固定位置提供商代替，如在TV设置中设置邮政编码。

```
// Request a static location from the location manager
LocationManager locationManager = (LocationManager) this.getSystemService(
        Context.LOCATION_SERVICE);
Location location = locationManager.getLastKnownLocation("static");

// Attempt to get postal or zip code from the static location object
Geocoder geocoder = new Geocoder(this);
Address address = null;
try {
  address = geocoder.getFromLocation(location.getLatitude(),
          location.getLongitude(), 1).get(0);
  Log.d("Zip code", address.getPostalCode());

} catch (IOException e) {
  Log.e(TAG, "Geocoder error", e);
}
```

处理控制器

TV设备需要辅助硬件设备与应用交互，如一个基本形式的遥控器或者游戏手柄。这意味着我们应用必须支持D-pad（十字方向键）输入。它也意味着我们应用可能需要处理手柄掉线和更多类型的手柄输入。

D-pad最低控制要求

默认的TV设备控制器是D-pad。通常，我们可以用遥控器的上，下，左，右，选择，返回，和Home键操作应用。如果应用是一个游戏而需要游戏手柄额外的控制，它也应该尝试允许用D-pad操作。这种情况下，应用也应该警告用户需要手柄，并且允许他们用D-pad优雅的退出游戏。更多关于在TV设备如理D-pad的操作，参考Creating TV Navigation。

处理手柄掉线

TV的手柄通常是蓝牙设备，它为了省电而定期的休眠并且与TV设备断开连接。这意味着如果不处理这些重连事件，应用可能被中断或者重新开始。这些事件可以发生在下面任何情景中：

- 当在看几分钟的视频，D-Pad或者游戏手柄进入了睡眠模式，从TV设备上断开连接并且随后重新连接。
- 在玩游戏时，新玩家用不是当前连接的游戏手柄加入游戏。
- 在玩游戏时，一个玩家离开游戏并且断开游戏手柄。

任何TV应用activity相关于断开和重连事件。这些事件必须在应用的manifest配置去处理。接下来的示例代码展示了如何确保一个activity去处理配置改变，包括键盘或者操作设备连接，断开连接，或者重新连接：

```
<activity
  android:name="com.example.android.TvActivity"
  android:label="@string/app_name"
  android:configChanges="keyboard|keyboardHidden|navigation"
  android:theme="@style/Theme.Leanback">

  <intent-filter>
    <action android:name="android.intent.action.MAIN" ></action>
    <category android:name="android.intent.category.LEANBACK_LAUNCHER" ></category>
  </intent-filter>
  ...
</activity>
```

这个配置改变属性允许应用通过重连事件继续运行，比较而言Android framework强制重启应用会导致一个不好的用户体验。

处理D-pad变种输入

TV设备用户可能有超过一种类型的控制器来操作TV。例如，一个用户可能有基本D-pad控制器和一个游戏控制器。游戏控制器用于D-pad功能的按键代码可能和物理十字键提供的不相同。

我们的应用应该处理游戏控制器D-pad的变种输入，这样用户不需要通过手动切换控制器去操作应用。更多信息关于处理这些变种输入，参考Handling Controller Actions。

下一节: 创建TV布局 >

创建TV布局

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/start/layouts.html

TV通常在3米外观看，并且它比大部分Android设备大的多。这类屏不能达到类似小设备的精细细节和颜色的水平。这些因素需要我们在头脑中考虑，并设计出对于TV设备更为有用且好用的应用布局。

这节课程描述了创建有效的TV应用布局的基本要求和实现细节。

用TV布局主题

Android主题能给我们的TV应用布局提供基础框架。对于打算在TV设备上运行的应用activity，我们应该用一款主题改变它的显示。这节课程教我们应该用哪个主题。

Leanback主题

支持TV用户界面的库叫做v17 leanback libarary，它提供了一个标准的TV activity主题，叫做 `Theme.Leanback` 。这一主题为TV应用程序建立了一致的视觉风格。强烈推荐在任何用了v17 leanback类的TV应用中使用这个主题。接下来的代码展示如何在应用中对给定的activity使用这个主题：

```
<activity
  android:name="com.example.android.TvActivity"
  android:label="@string/app_name"
  android:theme="@style/Theme.Leanback">
```

NoTitleBar主题

在手机和平板的Android应用中，标题栏是标准的用户界面元素。但是在TV应用中是不适合的。如果没有用v17 leanback类，我们应该在TV activity使用这个主题来隐去标题栏的显示。接下来的TV应用manifest代码示范了如何应用这个主题来删除标题栏。

```
<application>
  ...

  <activity
    android:name="com.example.android.TvActivity"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.NoTitleBar">
    ...

  </activity>
</application>
```

创建基本的TV布局

TV设备的布局应该遵循一些基本的指引确保它们在大屏幕下是可用的和有效率的。遵循这些技巧去创建最优化的TV横屏布局。

- 创建横屏布局。TV屏幕总是显示在横屏模式。
- 把导航控件放置在屏幕的左边或者右边，并且保持内容在垂直区间。
- 创建分离的UI，用Fragment，并且用框架如GridView代替ListView获得屏幕水平方向上更好的使用。
- 用框架如RelativeLayout或者LinearLayout来排列视图。基于对齐方式，纵横比，和电视屏幕的像素密度，这个方法允许系统调整视图大小的位置。
- 在布局控件之间添加足够的边际，以避免成为一个杂乱的UI。

Overscan

由于TV标准的演进，TV的布局有一个独特的需求是总是希望给观众显示全屏图像。因为这个原因，TV设备可能剪掉应用布局的外边缘去确保整个显示器被填满。这种行为通常简称为overscan。

避免屏幕元素由于overscan被剪掉，可以在布局所有的边缘增加总共10%的边际。这换算为在activity的基础布局上左右边缘留48dp的边际和在上下留27dp的边际。接下来的布局例子展示了如何在TV应用根布局上设置这些边际。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/base_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:layout_marginTop="27dp"
  android:layout_marginLeft="48dp"
  android:layout_marginRight="48dp"
  android:layout_marginBottom="27dp" >
</LinearLayout>
```

> **Caution**：如果我们正在使用v17 leanback类，不要在布局中留overscan边际，诸如BrowseFragment或者相关控件，因为那些布局已经包含了overscan安全边际。

创建方便使用的文本和控件

在TV应用布局中的文本和控件应该在一定距离外是容易查看和导航的。接下来的技巧是确保我们的用户界面元素在一定距离外更容易查看。

- 分解文本为小块，用户可以快速浏览。
- 在暗背景下用亮色文字。这种风格在TV中更容易阅读。
- 避免轻字体或者字体既窄且有非常宽阔的笔触效果。用简单的sans-serif字体并且去掉锯齿效果以增加可读性。
- 用Android标准的字体大小。

```
<TextView
      android:id="@+id/atext"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:gravity="center_vertical"
      android:singleLine="true"
      android:textAppearance="?android:attr/textAppearanceMedium"/>
```

- 确保所有的控件是足够大，使人们站在屏幕3米外（更大的屏幕这个距离会更大）可以看清楚。做这个最好的方式是用布局相对大小而不是绝对大小，并且用密度无关像素（dip）单位代替像素单位。例如，设置控件的宽度，用 `wrap_content` 代替特定像素值，并且设置控件的边际，用dip代替px值。 更多关于密度无关像素和创建大尺寸屏幕的布局，查看Support Mutiple Screens。

管理TV布局资源

通常的高清晰度TV分辨率是720p，1080i和1080p。假定我们的TV布局对象是一个1920 x 1080像素的屏幕，然后要允许Android系统必要情况下缩减布局元素到720p。通常，降低分辨率（删除像素）不会降低布局的外观质量。但是增加分辨率会降低布局显示的质量，并且会对用户体验造成负面影响。

为了获得最好的图像缩放效果，尽可能提供9-patch图片元素。如果在我们的布局中使用低质量或者小的图片，它们将出现马赛克，模糊或者颗粒，这不是一个好的用户体验。用高质量图片代替它。

更多关于优化布局和大屏幕的资源文件问题，参考Designing for multiple screens。

避免反模式布局

有几种创建布局的方法我们应该避免使用，因为它们不能在TV设备上很好的工作并且导致不好的用户体验。当开发TV布局时，以下一些用户界面是我们应该明确不能使用的。

- 重用手机和平板布局 - 不要重用没有修改的手机或者平板应用的布局。为其他Android设备开发的布局不适合TV设备，并且TV上布局应该被简化。
- 状态栏 - 尽管这种用户界面习惯是推荐使用在手机和平板上，但是他不适合TV界面。通常，状态栏选项菜单（或者任何下拉菜单）坚决不要使用，因为用遥控器操作这样的菜单是困难的。
- **ViewPager** - 在屏幕之间滑动能很好在手机或平板上工作，但是不要在TV上尝试！ 更多信息关于设计适合TV的布局，参考TV Design指导。

处理大图片

TV设备，像任何其他Android设备，内存有一定限制。如果我们创建的应用中用了很高分辨率的图片或者用了很多高分辨率图片，它可能很快达到内存限制，并且导致内存溢出错误。避免这些类型的问题，遵循以下方法：

- 仅当图片显示在屏幕时才加载。例如，当在GridView或者Gallery中显示多个图片时，仅当getView()在视图的Adapter中被调用时才加载图片。
- 在Bitmap视图中调用recycle()不再需要。
- 对存储在内存中集合中的位图对象使用弱引用。
- 如果我们从网络上获取图片，用AsyncTask去操作并且存储它们在设备上以方便更快的存取。绝对不要在应用的主线程操作网络传输。
- 当下载大图片时，降低图片到合适的尺寸，否则，下载图片本身可能导致内存溢出问题。 更多信息关于获得最好的图片操作性能，参考 Displaying Bitmaps Efficiently。

提供有效的广告

Android TV的广告必须总是全屏。广告不可以出现在内容的旁边或者覆盖内容。用户应当能用D-pad控制器关闭广告。视频广告在开始时间后的30秒内应当能被关闭。

Android TV不提供网页浏览器。我们的广告不应该尝试去启动网页浏览器或者重定向到Google Play商店。

> **Note**：WebView类用于登入服务器，如Google+和Facebook。

下一节: 创建TV导航 >

创建TV导航

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/start/navigation.html

TV设备为应用程序提供一组有限的导航控件。为我们的TV应用创建有效的导航方案取决于理解这些有限的控件和用户操作应用时的限制。因此当我们为TV创建Android应用时，额外注意用户是用遥控器按键,而不是用触摸屏导航我们的应用程序。

这节课解释了创建有效的TV应用导航方案的最低要求和如何对应用程序使用这些要求。

使用D-pad导航

在TV设备上，用户用遥控器设备的方向手柄（D-pad）或者方向键去控制控件。这类控制器限制为上下左右移动。为了创建最优化的TV应用，我们必须提供一个用户能快速学习如何使用有限控件导航的方案。

Android framework自动地处理布局元素之间的方向导航操作，因此我们不需要在应用中做额外的事情。不管怎样，我们也应该用D-pad控制器实际测试去发现任何导航问题。接下来的指引是如何在TV设备上用D-pad测试应用的导航。

- 确保用户能用D-pad控制器导航所有屏幕可见的控件。
- 对于滚动列表上的焦点，确保D-pad上下键能滚动列表，并且确定键能选择列表中的项。检查用户可以选择列表中的元素并且选中元素后仍可以滚动列表。
- 确保在控件之间切换是直接的和可预测的。

修改导航的方向

基于布局元素中可选中的元素的相对位置，Android framwork自动应用导航方向方案。我们应该用D-pad控制器测试生成的导航方案。在测试后，如果我们想规定用户以一个特定的方式在布局中移动，我们可以在控件中设置明确的导航方向。

> **Note**: 如果系统使用的默认顺序不是很好，我们应该仅用这些属性去修改导航顺序。

接下来的示例代码展示如何为TextView布局控件定义下一个控件焦点。

```
<TextView android:id="@+id/Category1"
        android:nextFocusDown="@+id/Category2"\>
```

接下来的列表展示了用户接口控件所有可用的导航属性。

属性	功能
nextFocusDown	定义用户按下导航时的焦点
nextFocusLeft	定义用户按左导航时的焦点
nextFocusRight	定义用户按右导航时的焦点
nextFocusUp	定义用户按上导航时的焦点

去使用这些明确的导航属性，设置另一个布局控件的ID值（ `android:id` 值）。我们应该设置导航顺序为一个循环，因此最后一个控件返回至第一个焦点。

提供清楚的焦点和选中状态

在TV设备上的应用导航方案的成功是基于用户如何容易的决定屏幕上界面元素的焦点。如果我们不提供清晰的焦点项显示（和用户能操作的选项），他们会很快泄气并退出我们的应用。同样的原因，重要的是当我们的应用开始或者任何无操作的时间中，总是有焦点项可以立即操作。

我们的应用布局和实现应该用颜色，大小，动画或者它们组在一起来帮助用户容易地决定下一步操作。在应用中用一致的焦点显示方案。

Android提供Drawable State List Resources来实现高亮选中的焦点。接下来的示例代码展示了如何为用户导航到控件并选择它时使用视觉化按钮显示：

```
<!-- res/drawable/button.xml -->
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /> <!-- pressed -->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /> <!-- focused -->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /> <!-- hovered -->
    <item android:drawable="@drawable/button_normal" /> <!-- default -->
</selector>
```

接下来的XML示例代码对按钮控件应用了上面的按键状态列表drawable：

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/button" />
```

确保在可定为焦点的和可选中的控件中提供了充分的填充，以便围绕它们的高亮是清楚的。

更多建议关于TV应用中设计有效的选中和焦点，看Patterns of TV。

下一节: 创建TV播放应用 >

创建TV播放应用

> 编写:huanglizhuo - 原文:http://developer.android.com/training/tv/playback/index.html

浏览和播放媒体文件往往是由一个TV应用程序提供的用户体验的一部分。从头开始构建这样的体验，并同时确保它是快速，流畅，和有吸引力的是具有相当挑战性的。您的应用程序提供访问媒体类别无论大小，允许用户快速浏览选项，并获得他们想要的内容是很重要的。

Android框架通过v17 leanback support library为构建用户界面提供接口。该库提供类来创建用于浏览和播放多媒体的高效框架,为开发者减少代码。该类可以进行扩展和定制，所以我们可以为我们的应用程序创建一个独特的高效的类。

这节课将向您介绍如何用Leanback的支持库构建用于浏览和播放TV媒体内容的TV应用程序。

主题

- 创建一个类别浏览器

  学习如何使用Leanback的支持库，建立一个媒体类别的浏览界面。

- 提供一个卡片**View**

  学习使用Leanback的支持库，建立一个卡片视图的内容项目。

- 创建详细信息**View**

  学习使用Leanback的支持库，建立一个详细内容展示页。

- 显示正在播放卡片

  学习如何使用MediaSession在主屏幕上显示正在播放。

下一节：创建一个类别浏览器 >

创建目录浏览器

> 编写:huanglizhuo - 原文:http://developer.android.com/training/tv/playback/browse.html

在TV上运行的多媒体应用得允许用户浏览,选择和播放它所提供的内容。目录浏览器的用户体验要简单和直观，以及赏心悦目，引人入胜。

这节课讨论如何使用的V17 Leanback库提供的类来实现用户界面，用于从您的应用程序的媒体目录浏览音乐或视频。

创建一个目录布局

leanback 类库中的BrowseFragment允许您用最少的代码创建一个用于按行浏览的主布局 ,下面的例子将演示如何创建包含BrowseFragment的布局

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  >

  <fragment
      android:name="android.support.v17.leanback.app.BrowseFragment"
      android:id="@+id/browse_fragment"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      />
</LinearLayout>
```

为了使 activity 工作,需要在布局中取回BrowseFragment的元素。使用这个类中的方法设置显示参数,如图标,标题,以及该类别是否可用。下面的代码简单的演示了怎样设置BrowseFragment布局参数:

```
public class BrowseMediaActivity extends Activity {

    public static final String TAG ="BrowseActivity";

    protected BrowseFragment mBrowseFragment;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.browse_fragment);

        final FragmentManager fragmentManager = getFragmentManager();
        mBrowseFragment = (BrowseFragment) fragmentManager.findFragmentById(
                R.id.browse_fragment);

        // Set display parameters for the BrowseFragment
        mBrowseFragment.setHeadersState(BrowseFragment.HEADERS_ENABLED);
        mBrowseFragment.setTitle(getString(R.string.app_name));
        mBrowseFragment.setBadgeDrawable(getResources().getDrawable(
                R.drawable.ic_launcher));
        mBrowseFragment.setBrowseParams(params);

    }
}
```

显示媒体列表

BrowseFragment允许您定义和使用 adapter 和presenter 定义显示可浏览媒体内容类别和媒体项目。Adapters 允许我们连接本地或网络数据资源。Presenters操控的媒体项目的数据，并提供布局信息在屏幕上显示的项目。

下面的示例代码演示了一个为显示字符串数据的Presenters的实现

```
public class StringPresenter extends Presenter {
    private static final String TAG = "StringPresenter";

    public ViewHolder onCreateViewHolder(ViewGroup parent) {
        TextView textView = new TextView(parent.getContext());
        textView.setFocusable(true);
        textView.setFocusableInTouchMode(true);
        textView.setBackground(
                parent.getContext().getResources().getDrawable(R.drawable.text_bg));
        return new ViewHolder(textView);
    }

    public void onBindViewHolder(ViewHolder viewHolder, Object item) {
        ((TextView) viewHolder.view).setText(item.toString());
    }

    public void onUnbindViewHolder(ViewHolder viewHolder) {
        // no op
    }
}
```

当我们已经为我们的媒体项目构建了一个Presenter类，我们可以为BrowseFragment建立并添加一个适配器并在屏幕上显示这些媒体项目。下面的示例代码演示了如何用StringPresenter类构造一个类别和项目适配器：

```
private ArrayObjectAdapter mRowsAdapter;
private static final int NUM_ROWS = 4;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    buildRowsAdapter();
}

private void buildRowsAdapter() {
    mRowsAdapter = new ArrayObjectAdapter(new ListRowPresenter());

    for (int i = 0; i < NUM_ROWS; ++i) {
        ArrayObjectAdapter listRowAdapter = new ArrayObjectAdapter(
                new StringPresenter());
        listRowAdapter.add("Media Item 1");
        listRowAdapter.add("Media Item 2");
        listRowAdapter.add("Media Item 3");
        HeaderItem header = new HeaderItem(i, "Category " + i, null);
        mRowsAdapter.add(new ListRow(header, listRowAdapter));
    }

    mBrowseFragment.setAdapter(mRowsAdapter);
}
```

这个例子显示了静态实现适配器。典型的媒体浏览器使用网络数据库或网络服务。使用从网络取回的数据做的媒体浏览器,参看例子Android TV

更新背景

为了给媒体浏览应用增加视觉趣味，我们可以在用户浏览的内容时更新背景图片。这种技术可以让我们的应用程序的互动感倍增。

Leanback库提供了BackgroundManager类为我们的TV应用的activity更换背景。下面的例子演示了如何创建一个简单的方法更换背景:

```
protected void updateBackground(Drawable drawable) {
    BackgroundManager.getInstance(this).setDrawable(drawable);
}
```

许多现有的媒体浏览应用在用户浏览媒体列表自动更新的背景。为了做到这一点，我们可以设置一个选择监听器，根据用户的当前选择自动更新背景。下面的例子演示了如何建立一个OnItemViewSelectedListener监听选择事件并更新背景：

```
protected void clearBackground() {
    BackgroundManager.getInstance(this).setDrawable(mDefaultBackground);
}

protected OnItemViewSelectedListener getDefaultItemViewSelectedListener() {
    return new OnItemViewSelectedListener() {
        @Override
        public void onItemSelected(Object item, Row row) {
            if (item instanceof Movie ) {
                URI uri = ((Movie)item).getBackdropURI();
                updateBackground(uri);
            } else {
                clearBackground();
            }
        }
    };
}
```

> 注意:以上的示例是为了简单。当我们在自己的应用程序创建这个功能，我们应该考虑运行在一个单独的线程在后台更新操作获得更好的性能。此外，如果我们正计划在用户触发项目滚动时更新背景，考虑增加一个时延，直到用户停止操作时再更新背景图像。这样可以避免过多的背景图片的更新。

下一节：提供一个卡片View >

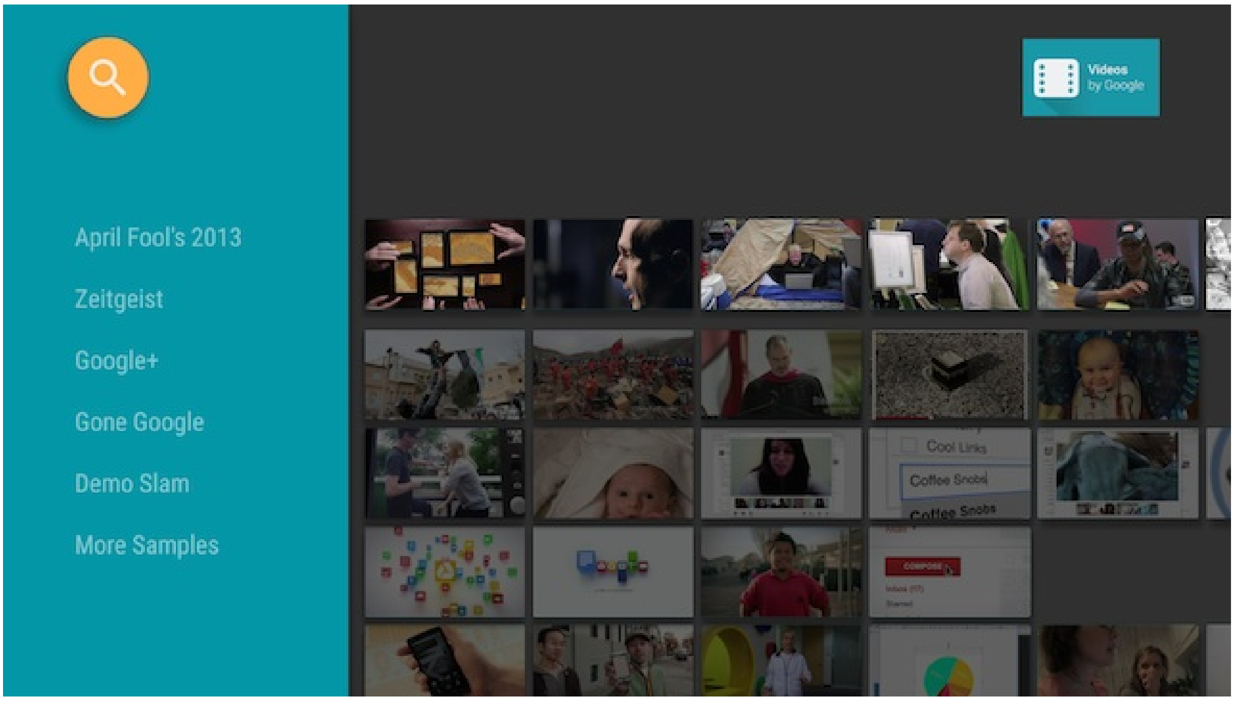
提供一个Card视图

编写:huanglizhuo - 原文:http://developer.android.com/training/tv/playback/card.html

在前面的课程中，我们创建一个目录浏览器，实现了浏览 fragment，显示了媒体项目的列表。在本课程中，我们将创建该卡视图的媒体项目，并在浏览fragment中呈现出来。

BaseCardView类以及子类显示与媒体项目相关联的元数据。在本节课程中使用的ImageCardView类显示随着媒体项目的标题内容的图像。

这节课介绍了GitHub上 Android Leanback sample app的示例应用程序代码。使用该示例代码，开始我们自己的应用程序。



创建一个卡片呈现者

Presenter生成视图并把类和它们绑定起来。在我们的浏览 fragment 中将内容呈现给用户,我们为内容卡片创建Presenter并把它传给适配器然后将内容呈现在屏幕上。在下面的代码中,CardPresenter在 LoaderManager的onLoadFinished)方法中被创建。

```
@Override
public void onLoadFinished(Loader<HashMap<String, List<Movie>>> arg0,
                           HashMap<String, List<Movie>> data) {

    mRowsAdapter = new ArrayObjectAdapter(new ListRowPresenter());
    CardPresenter cardPresenter = new CardPresenter();

    int i = 0;

    for (Map.Entry<String, List<Movie>> entry : data.entrySet()) {
        ArrayObjectAdapter listRowAdapter = new ArrayObjectAdapter(cardPresenter);
        List<Movie> list = entry.getValue();

        for (int j = 0; j < list.size(); j++) {
            listRowAdapter.add(list.get(j));
        }
        HeaderItem header = new HeaderItem(i, entry.getKey(), null);
        i++;
        mRowsAdapter.add(new ListRow(header, listRowAdapter));
    }

    HeaderItem gridHeader = new HeaderItem(i, getString(R.string.more_samples),
            null);

    GridItemPresenter gridPresenter = new GridItemPresenter();
    ArrayObjectAdapter gridRowAdapter = new ArrayObjectAdapter(gridPresenter);
    gridRowAdapter.add(getString(R.string.grid_view));
    gridRowAdapter.add(getString(R.string.error_fragment));
    gridRowAdapter.add(getString(R.string.personal_settings));
    mRowsAdapter.add(new ListRow(gridHeader, gridRowAdapter));

    setAdapter(mRowsAdapter);

    updateRecommendations();
}
```

创建一个卡片视图

在这步中,我们将用view holder创建一个卡片presenter来为卡片视图呈现媒体项目。注意,每个presenter只能创建一个view类别。如果我们有俩个不同的卡片视图,我们就得创建俩个不同的presenter

在presenter实现onCreateViewHolder)时创建一个可以呈现内容项目的view holder。

```
@Override
public class CardPresenter extends Presenter {

    private Context mContext;
    private static int CARD_WIDTH = 313;
    private static int CARD_HEIGHT = 176;
    private Drawable mDefaultCardImage;

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent) {
        mContext = parent.getContext();
        mDefaultCardImage = mContext.getResources().getDrawable(R.drawable.movie);
...
```

在onCreateViewHolder)方法中,创建呈现内容的卡片视图。下面的例子用的是ImageCardView

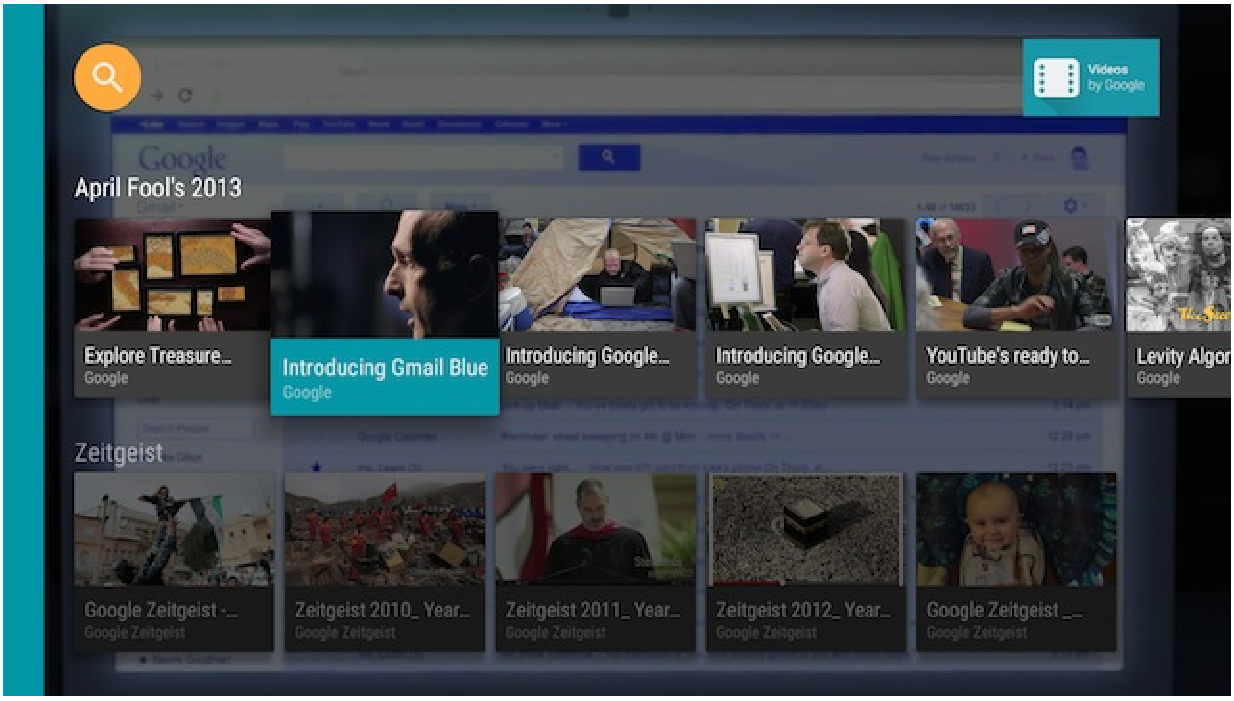
当卡片被选中时,默认的行为是放大展开。如果我们想创建不同颜色的卡片可以向下面这样调用setSelected)方法中实现。

```
...
    ImageCardView cardView = new ImageCardView(mContext) {
        @Override
        public void setSelected(boolean selected) {
            int selected_background = mContext.getResources().getColor(R.color.detail_
background);
            int default_background = mContext.getResources().getColor(R.color.default_
background);
            int color = selected ? selected_background : default_background;
            findViewById(R.id.info_field).setBackgroundColor(color);
            super.setSelected(selected);
        }
    };
...
```

当用户打开我们的应用时,Presenter.ViewHolder 为内容项目显示了卡片视图。我们需要调用setFocusable(true))和setFocusableInTouchMode(true))方法设置接收来自D-pad的焦点控制。

```
...
    cardView.setFocusable(true);
    cardView.setFocusableInTouchMode(true);
    return new ViewHolder(cardView);
}
```

当用户选中ImageCardView时,它用我们制定的颜色背景展开文字内容,就像下面这样。



下一节：创建详细信息View >

创建详情页

编写:huanglizhuo - 原文:http://developer.android.com/training/tv/playback/details.html

待认领进行编写，有意向的小伙伴，可以直接修改对应的markdown文件，进行提交！

v17 leanback support library 库提供的媒体浏览接口包含显示附加媒体信息的类,比如描述和预览,以及对项目的操作,比如购买或播放。

这节课讨论如何为媒体项目的详细信息创建 presenter 类，以及用户选择一个媒体项目时如何扩展 DetailsFragment类来实现显示媒体详细信息视图。

> 小贴士: 这里的实现例子用的是包含 DetailsFragment的附加activity。但也可以在同一个 activity 中用 fragment 转换将 BrowseFragment替换为 DetailsFragment.更多关于 fragment的信息请参考Building a Dynamic UI with Fragments

创建详细Presenter

在leanback库提供的媒体浏览框架中,可以用presenter对象控制屏幕显示数据,包括媒体详细信息。AbstractDetailsDescriptionPresenter 类提供的框架几乎是媒体项目详细信息的完全继承。我们只需要实现onBindDescription()方法,像下面这样把数据信息和视图绑定起来。

```
public class DetailsDescriptionPresenter
        extends AbstractDetailsDescriptionPresenter {

    @Override
    protected void onBindDescription(ViewHolder viewHolder, Object itemData) {
        MyMediaItemDetails details = (MyMediaItemDetails) itemData;
        // In a production app, the itemData object contains the information
        // needed to display details for the media item:
        // viewHolder.getTitle().setText(details.getShortTitle());

        // Here we provide static data for testing purposes:
        viewHolder.getTitle().setText(itemData.toString());
        viewHolder.getSubtitle().setText("2014   Drama   TV-14");
        viewHolder.getBody().setText("Lorem ipsum dolor sit amet, consectetur "
                + "adipisicing elit, sed do eiusmod tempor incididunt ut labore "
                + " et dolore magna aliqua. Ut enim ad minim veniam, quis "
                + "nostrud exercitation ullamco laboris nisi ut aliquip ex ea "
                + "commodo consequat.");
    }
}
```

扩展详细fragment

当使用 DetailsFragment类显示我们的媒体项目详细信息时,扩展该类并提供像预览图片,操作等附加内容。我们也可以提供一系列的相关媒体信息。

下面的例子演示了怎样用presenter类为媒体项目添加预览图片和操作。这个例子也演示了添加相关媒体行。

```
public class MediaItemDetailsFragment extends DetailsFragment {
    private static final String TAG = "MediaItemDetailsFragment";
    private ArrayObjectAdapter mRowsAdapter;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        Log.i(TAG, "onCreate");
        super.onCreate(savedInstanceState);

        buildDetails();
    }

    private void buildDetails() {
        ClassPresenterSelector selector = new ClassPresenterSelector();
        // Attach your media item details presenter to the row presenter:
        DetailsOverviewRowPresenter rowPresenter =
            new DetailsOverviewRowPresenter(new DetailsDescriptionPresenter());

        selector.addClassPresenter(DetailsOverviewRow.class, rowPresenter);
        selector.addClassPresenter(ListRow.class,
                new ListRowPresenter());
        mRowsAdapter = new ArrayObjectAdapter(selector);

        Resources res = getActivity().getResources();
        DetailsOverviewRow detailsOverview = new DetailsOverviewRow(
                "Media Item Details");

        // Add images and action buttons to the details view
        detailsOverview.setImageDrawable(res.getDrawable(R.drawable.jelly_beans));
        detailsOverview.addAction(new Action(1, "Buy $9.99"));
        detailsOverview.addAction(new Action(2, "Rent $2.99"));
        mRowsAdapter.add(detailsOverview);

        // Add a Related items row
        ArrayObjectAdapter listRowAdapter = new ArrayObjectAdapter(
                new StringPresenter());
        listRowAdapter.add("Media Item 1");
        listRowAdapter.add("Media Item 2");
        listRowAdapter.add("Media Item 3");
        HeaderItem header = new HeaderItem(0, "Related Items", null);
        mRowsAdapter.add(new ListRow(header, listRowAdapter));

        setAdapter(mRowsAdapter);
    }
}
```

创建详细信息activity

像 DetailsFragment这样的 fragment 为了使用或显示必须包含activity。为我们的详细信息与浏览分开创建activity并通过传递Intent打开。这节演示了如何创建一个包含媒体详细信息的activity。

创建详细信息前先为 DetailsFragment创建一个布局文件:

```
<!-- file: res/layout/details.xml -->

<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.example.android.mediabrowser.MediaItemDetailsFragment"
    android:id="@+id/details_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

接下来用上面的布局文件创建一个activity:

```
public class DetailsActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.details);
    }
}
```

最后在manifest文件中申明activity。记得添加Leanback主题以确保用户界面中有媒体浏览activity。

```
<application>
  ...

  <activity android:name=".DetailsActivity"
    android:exported="true"
    android:theme="@style/Theme.Leanback"/>

</application>
```

为点击项目添加Listener

实现 DetailsFragment后,在用户点击媒体条目时将我们的媒体浏览view切换详细信息view。为了确保动作的实现,在BrowserFragment中添加[OnItemViewClickedListener]通过Intent开启详细信息activity。

下面的例子演示了实现怎样在媒体浏览view中实现一个 listener开启详细信息view。

```
public class BrowseMediaActivity extends Activity {
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...

        // create the media item rows
        buildRowsAdapter();

        // add a listener for selected items
        mBrowseFragment.OnItemViewClickedListener(
            new OnItemViewClickedListener() {
                @Override
                public void onItemClicked(Object item, Row row) {
                    System.out.println("Media Item clicked: " + item.toString());
                    Intent intent = new Intent(BrowseMediaActivity.this,
                            DetailsActivity.class);
                    // pass the item information
                    intent.getExtras().putLong("id", item.getId());
                    startActivity(intent);
                }
            });
    }
}
```

下一节：显示正在播放卡片 >

显示正在播放卡片

> 编写:huanglizhuo - 原文:http://developer.android.com/training/tv/playback/now-playing.html

TV应用允许用户在使用其他应用时后台播放音乐或其他媒体。如果我们的应用程序允许后台，它必须要为用户提供返回该应用暂停音乐或切换到一个新的歌曲的方法。 Android框架允许TV应用通过在主屏幕上显示正在播放卡做到这一点。

正在播放卡片是系统的组建,它可以在推荐的行上显示正在播放的媒体会话它包括了媒体元数据，如专辑封面，标题和应用程序图标。当用户选择它，系统将打开拥有该会话的应用程序。

这节课将演示如何使用 MediaSession 类实现正在播放卡片。

开启媒体会话

一个播放应用可以作为 activity 或者 service 运行。 service 是当 activity 结束时依然可以后台播放的。在这节讨论中,媒体播放应用是假设在 MediaBrowserService 下运行的。

在service的onCreate())方法中创建一个新的 MediaSession),设置适当的回调函数和标志,并设置 MediaBrowserService 令牌。

```
mSession = new MediaSession(this, "MusicService");
mSession.setCallback(new MediaSessionCallback());
mSession.setFlags(MediaSession.FLAG_HANDLES_MEDIA_BUTTONS |
        MediaSession.FLAG_HANDLES_TRANSPORT_CONTROLS);

// for the MediaBrowserService
setSessionToken(mSession.getSessionToken());
```

> 注意:正在播放卡片只有在媒体会话设置了FLAG_HANDLES_TRANSPORT_CONTROLS标志时在可以显示。

显示正在播放卡片

如果会话是系统最高优先级的会话那么正在播放卡片将在setActivity(true))调用后显示。同时我们的应用必须像在Managing Audio Focus一节中那样请求音频焦点。

```
private void handlePlayRequest() {

    tryToGetAudioFocus();

    if (!mSession.isActive()) {
        mSession.setActive(true);
    }
...
```

如果另一个应用发起媒体播放请求并调用setActivity(false))后这个卡片将从主屏上移除。

更新播放状态

正如任何媒体的应用程序，在 MediaSession 中更新播放状态，使卡片可以显示当前的元数据，如在下面的例子：

```
private void updatePlaybackState() {
    long position = PlaybackState.PLAYBACK_POSITION_UNKNOWN;
    if (mMediaPlayer != null && mMediaPlayer.isPlaying()) {
        position = mMediaPlayer.getCurrentPosition();
    }
    PlaybackState.Builder stateBuilder = new PlaybackState.Builder()
            .setActions(getAvailableActions());
    stateBuilder.setState(mState, position, 1.0f);
    mSession.setPlaybackState(stateBuilder.build());
}
private long getAvailableActions() {
    long actions = PlaybackState.ACTION_PLAY |
            PlaybackState.ACTION_PLAY_FROM_MEDIA_ID |
            PlaybackState.ACTION_PLAY_FROM_SEARCH;
    if (mPlayingQueue == null || mPlayingQueue.isEmpty()) {
        return actions;
    }
    if (mState == PlaybackState.STATE_PLAYING) {
        actions |= PlaybackState.ACTION_PAUSE;
    }
    if (mCurrentIndexOnQueue > 0) {
        actions |= PlaybackState.ACTION_SKIP_TO_PREVIOUS;
    }
    if (mCurrentIndexOnQueue < mPlayingQueue.size() - 1) {
        actions |= PlaybackState.ACTION_SKIP_TO_NEXT;
    }
    return actions;
}
```

显示媒体元数据

为当前正在播放通过setMetadata())方法设置 MediaMetadata 。.这个方法可以让我们为正在播放卡提供有关轨道，如标题，副标题，和各种图标等信息。下面的例子假设我们的播放数据存储在自定义的MediaData类中。

```
private void updateMetadata(MediaData myData) {
    MediaMetadata.Builder metadataBuilder = new MediaMetadata.Builder();
    // To provide most control over how an item is displayed set the
    // display fields in the metadata
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_TITLE,
            myData.displayTitle);
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_SUBTITLE,
            myData.displaySubtitle);
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_ICON_URI,
            myData.artUri);
    // And at minimum the title and artist for legacy support
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_TITLE,
            myData.title);
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_ARTIST,
            myData.artist);
    // A small bitmap for the artwork is also recommended
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_ART,
            myData.artBitmap);
    // Add any other fields you have for your data as well
    mSession.setMetadata(metadataBuilder.build());
}
```

响应用户的动作

当用户选择正在播放卡片时,系统打开应用并拥有会话。如果我们的应用在setSessionActivity())有PendingIntent要传递,系统将会像下面演示的那样开启activity。如果不是，则系统默认的Intent打开。您指定的活动必须提供播放控制，允许用户暂停或停止播放。

```
Intent intent = new Intent(mContext, MyActivity.class);
    PendingIntent pi = PendingIntent.getActivity(context, 99 /*request code*/,
            intent, PendingIntent.FLAG_UPDATE_CURRENT);
    mSession.setSessionActivity(pi);
```

帮助用户在TV上找到内容

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/discovery/index.html

TV设备为用户提供了许多的休闲娱乐选择。它们提供上千个应用和相关的内容服务。同时，大部分用户操作TV时，喜欢比较少的输入操作。面对用户可能的选择，重要的一点是应用开发者为用户提供快速容易的路径，发现和享受我们的内容。

Android framework层帮助我们为用户提供若干路径，去找到内容，包括主屏幕的推荐和应用的内容目录的搜索。

这节课展示如何帮助用户找到应用内容，通过推荐和应用内搜索。

主题

- 推荐TV内容 学习如何推荐内容给用户，使它出现在TV设备的主屏幕推荐栏。
- 使TV应用是可被搜索的 学习如何使内容在Android TV主屏幕中被搜索到。
- TV应用内搜索 学习如何在应用内使用内置的TV搜索界面。

推荐TV内容 >

推荐TV内容

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/discovery/recommendations.html

当操作TV时，用户通常喜欢使用最少的输入操作来找内容。许多用户的理想场景是，坐下，打开TV然后观看。用最少的步骤让用户观看他们的喜欢的内容是最好的方式。

Android framework为了实现较少交互而提供了主屏幕推荐栏。在设备第一次使用时候，内容推荐出现在TV主屏幕的第一栏。应用程序的内容目录提供推荐建议可以把用户带回到我们的应用。

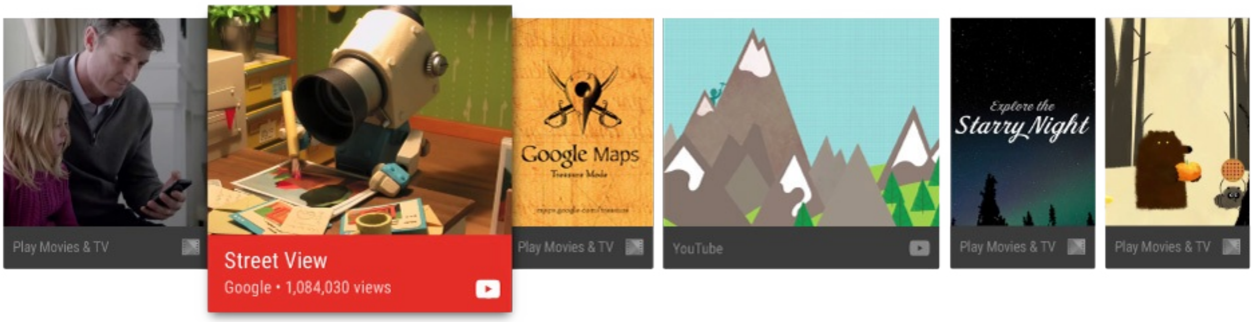


图1. 一个推荐栏的例子

这节课教我们如何创建推荐和提供他们到Android framework，这样用户能容易的发现和使用我们的应用内容。这个讨论描述了一些代码，在Android Leanback示例代码。

创建推荐服务

内容推荐是被后台处理创建。为了把我们的应用提供到内容推荐，创建一个周期性添加列表服务，从应用目录到系统推荐列表。

接下来的代码描绘了如何扩展IntentService为我们的应用创建推荐服务：

```
public class UpdateRecommendationsService extends IntentService {
    private static final String TAG = "UpdateRecommendationsService";
    private static final int MAX_RECOMMENDATIONS = 3;

    public UpdateRecommendationsService() {
        super("RecommendationService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
```

```
        Log.d(TAG, "Updating recommendation cards");
        HashMap<String, List<Movie>> recommendations = VideoProvider.getMovieList();
        if (recommendations == null) return;

        int count = 0;

        try {
            RecommendationBuilder builder = new RecommendationBuilder()
                    .setContext(getApplicationContext())
                    .setSmallIcon(R.drawable.videos_by_google_icon);

            for (Map.Entry<String, List<Movie>> entry : recommendations.entrySet()) {
                for (Movie movie : entry.getValue()) {
                    Log.d(TAG, "Recommendation - " + movie.getTitle());

                    builder.setBackground(movie.getCardImageUrl())
                            .setId(count + 1)
                            .setPriority(MAX_RECOMMENDATIONS - count)
                            .setTitle(movie.getTitle())
                            .setDescription(getString(R.string.popular_header))
                            .setImage(movie.getCardImageUrl())
                            .setIntent(buildPendingIntent(movie))
                            .build();

                    if (++count >= MAX_RECOMMENDATIONS) {
                        break;
                    }
                }
                if (++count >= MAX_RECOMMENDATIONS) {
                    break;
                }
            }
        } catch (IOException e) {
            Log.e(TAG, "Unable to update recommendation", e);
        }
    }

    private PendingIntent buildPendingIntent(Movie movie) {
        Intent detailsIntent = new Intent(this, DetailsActivity.class);
        detailsIntent.putExtra("Movie", movie);

        TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
        stackBuilder.addParentStack(DetailsActivity.class);
        stackBuilder.addNextIntent(detailsIntent);
        // Ensure a unique PendingIntents, otherwise all recommendations end up with t
he same
        // PendingIntent
        detailsIntent.setAction(Long.toString(movie.getId()));

        PendingIntent intent = stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPD
ATE_CURRENT);
        return intent;
    }
```

```
}
```

使服务被系统意识和运行，在应用manifest中注册它，接下来的代码片段展示了如何定义这个类做为服务：

```
<manifest ... >
  <application ... >
    ...

    <service
            android:name="com.example.android.tvleanback.UpdateRecommendationsService"
            android:enabled="true" />
  </application>
</manifest>
```

刷新推荐

基于用户的行为和数据来推荐，例如播放列表，喜爱列表和相关内容。当刷新推荐时，不仅仅是删除和重新加载他们，因为这样会导致推荐出现在推荐栏的结尾。一旦一个内容项被播放，如一个影片，从推荐中删除它。

应用的推荐顺序被保存依据应用提供他们的顺序。framework interleave应用推荐基于推荐质量，用户习惯的收集。最好的推荐应是推荐最合适的出现在列表前面。

创建推荐

一旦我们的推荐服务开始运行，它必须创建推荐和推送他们到Android framework。Framework收到推荐作为通知对象。它用特定的模板并且标记为特定的目录。

设置值

去设置推荐卡片的UI元素，创建一个builder类用接下来的builder样式描述。首先，设置推荐卡片元素的值。

```
public class RecommendationBuilder {
    ...

    public RecommendationBuilder setTitle(String title) {
            mTitle = title;
            return this;
        }

        public RecommendationBuilder setDescription(String description) {
            mDescription = description;
            return this;
        }

        public RecommendationBuilder setImage(String uri) {
            mImageUri = uri;
            return this;
        }

        public RecommendationBuilder setBackground(String uri) {
            mBackgroundUri = uri;
            return this;
        }
...
```

创建通知

一旦我们设置了值，然后去创建通知，从builder类分配值到通知，并且调用NotificationCompat.Builder.build)。

并且，确信调用setLocalOnly())，这样NotificationCompat.BigPictureStyle通知不将显示在另一个设备。

接下来的代码示例展示了如何创建推荐。

```
public class RecommendationBuilder {
    ...

    public Notification build() throws IOException {
        ...

        Notification notification = new NotificationCompat.BigPictureStyle(
                new NotificationCompat.Builder(mContext)
                        .setContentTitle(mTitle)
                        .setContentText(mDescription)
                        .setPriority(mPriority)
                        .setLocalOnly(true)
                        .setOngoing(true)
                        .setColor(mContext.getResources().getColor(R.color.fastlane_ba
ckground))
                        .setCategory(Notification.CATEGORY_RECOMMENDATION)
                        .setLargeIcon(image)
                        .setSmallIcon(mSmallIcon)
                        .setContentIntent(mIntent)
                        .setExtras(extras))
                .build();

        return notification;
    }
}
```

运行推荐服务

我们的应用推荐服务必须周期性运行确保创建当前的推荐。去运行我们的服务，创建一个类运行计时器和在周期间隔关联它。接下来的代码例子扩展了BroadcastReceiver类去开始每半小时的推荐服务的周期性执行：

```
public class BootupActivity extends BroadcastReceiver {
    private static final String TAG = "BootupActivity";

    private static final long INITIAL_DELAY = 5000;

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "BootupActivity initiated");
        if (intent.getAction().endsWith(Intent.ACTION_BOOT_COMPLETED)) {
            scheduleRecommendationUpdate(context);
        }
    }

    private void scheduleRecommendationUpdate(Context context) {
        Log.d(TAG, "Scheduling recommendations update");

        AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.AL
ARM_SERVICE);
        Intent recommendationIntent = new Intent(context, UpdateRecommendationsService
.class);
        PendingIntent alarmIntent = PendingIntent.getService(context, 0, recommendatio
nIntent, 0);

        alarmManager.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                INITIAL_DELAY,
                AlarmManager.INTERVAL_HALF_HOUR,
                alarmIntent);
    }
}
```

这个BroadcastReceiver类的实现必须运行在TV设备启动后。 为了完成这个，注册这个类在应用manifest的intet filter中，它监听设备启动完成。接下来的代码展示了如何添加这个配置到manifest。

```
<manifest ... >
  <application ... >
    <receiver android:name="com.example.android.tvleanback.BootupActivity"
              android:enabled="true"
              android:exported="false">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

> **Important**： 接收一个启动完成通知需要我们的应用有RECEIVE_BOOT_COMPLETED权限。更多信息，查看ACTION_BOOT_COMPLETED。

在推荐服务类的onHandleIntent())方法中，用以下代码提交推荐到管理器：

```
Notification notification = notificationBuilder.build();
mNotificationManager.notify(id, notification);
```

下一节: 使TV应用是可被搜索的 >

使TV应用是可被搜索的

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/discovery/searchable.html

Android TV使用Android搜索接口从安装的应用中检索内容数据并且释放搜索结果给用户。我们的应用内容数据能被包含在这些结果中，去给用户即时访问应用程序中的内容。

我们的应用必须提供Android TV数据字段，它是用户在搜索框中输入字符生成的建议搜索结果。去做这个，我们的应用必须实现Content Provider，在searchable.xml配置文件描述content provider和其他必要的Android TV信息。我们也需要一个activity在用户选择一个建议的搜索结果时处理intent的触发。所有的这些被描述在Adding Custom Suggestions。本文描述Android TV应用搜索的关键点。

这节课展示Android中搜索的知识，展示如何使我们的应用在Android TV里是可被搜索的。确信我们熟悉Search API guide的解释。在下面的这节课程之前，查看Adding Search Functionality训练课程。

这个讨论描述的一些代码，从Android Leanback示例代码摘出。代码可以在Github上找到。

识别列

SearchManager描述了数据字段，它被代表为SQLite数据库的列。不管我们的数据格式，我们必须把我们的数据字段填到那些列，通常用存取我们的内容数据的类。更多信息，查看Building a suggestion table()。

SearchManager类为AndroidTV包含了几个列。下面是重要的一些列：

值	描述
`SUGGEST_COLUMN_TEXT_1`	内容名字 (必须)
`SUGGEST_COLUMN_TEXT_2`	内容的文本描述
`SUGGEST_COLUMN_RESULT_CARD_IMAGE`	图片/封面
`SUGGEST_COLUMN_CONTENT_TYPE`	媒体的MIME类型 (必须)
`SUGGEST_COLUMN_VIDEO_WIDTH`	媒体的分辨率宽度
`SUGGEST_COLUMN_VIDEO_HEIGHT`	媒体的分辨率高度
`SUGGEST_COLUMN_PRODUCTION_YEAR`	内容的产品年份 (必须)
`SUGGEST_COLUMN_DURATION`	媒体的时间长度

搜索framework需要以下的列：

- SUGGEST_COLUMN_TEXT_1
- SUGGEST_COLUMN_CONTENT_TYPE
- SUGGEST_COLUMN_PRODUCTION_YEAR

当这些内容的列的值匹配Google服务的providers提供的的值时，系统提供一个深链接到我们的应用，用于详情查看，以及指向应用的其他Providers的链接。更多讨论在在详情页显示内容。

我们的应用的数据库类可能定义以下的列：

```
public class VideoDatabase {
  //The columns we'll include in the video database table
  public static final String KEY_NAME = SearchManager.SUGGEST_COLUMN_TEXT_1;
  public static final String KEY_DESCRIPTION = SearchManager.SUGGEST_COLUMN_TEXT_2;
  public static final String KEY_ICON = SearchManager.SUGGEST_COLUMN_RESULT_CARD_IMAGE
;
  public static final String KEY_DATA_TYPE = SearchManager.SUGGEST_COLUMN_CONTENT_TYPE
;
  public static final String KEY_IS_LIVE = SearchManager.SUGGEST_COLUMN_IS_LIVE;
  public static final String KEY_VIDEO_WIDTH = SearchManager.SUGGEST_COLUMN_VIDEO_WIDT
H;
  public static final String KEY_VIDEO_HEIGHT = SearchManager.SUGGEST_COLUMN_VIDEO_HEI
GHT;
  public static final String KEY_AUDIO_CHANNEL_CONFIG =
          SearchManager.SUGGEST_COLUMN_AUDIO_CHANNEL_CONFIG;
  public static final String KEY_PURCHASE_PRICE = SearchManager.SUGGEST_COLUMN_PURCHAS
E_PRICE;
  public static final String KEY_RENTAL_PRICE = SearchManager.SUGGEST_COLUMN_RENTAL_PR
ICE;
  public static final String KEY_RATING_STYLE = SearchManager.SUGGEST_COLUMN_RATING_ST
YLE;
  public static final String KEY_RATING_SCORE = SearchManager.SUGGEST_COLUMN_RATING_SC
ORE;
  public static final String KEY_PRODUCTION_YEAR = SearchManager.SUGGEST_COLUMN_PRODUC
TION_YEAR;
  public static final String KEY_COLUMN_DURATION = SearchManager.SUGGEST_COLUMN_DURATI
ON;
  public static final String KEY_ACTION = SearchManager.SUGGEST_COLUMN_INTENT_ACTION;
...
```

当我们创建从SearchManager列填充到我们的数据字段时，我们也必须定义_ID去获得每行的独一无二的ID。

```
...
  private static HashMap buildColumnMap() {
    HashMap map = new HashMap();
    map.put(KEY_NAME, KEY_NAME);
    map.put(KEY_DESCRIPTION, KEY_DESCRIPTION);
    map.put(KEY_ICON, KEY_ICON);
    map.put(KEY_DATA_TYPE, KEY_DATA_TYPE);
    map.put(KEY_IS_LIVE, KEY_IS_LIVE);
    map.put(KEY_VIDEO_WIDTH, KEY_VIDEO_WIDTH);
    map.put(KEY_VIDEO_HEIGHT, KEY_VIDEO_HEIGHT);
    map.put(KEY_AUDIO_CHANNEL_CONFIG, KEY_AUDIO_CHANNEL_CONFIG);
    map.put(KEY_PURCHASE_PRICE, KEY_PURCHASE_PRICE);
    map.put(KEY_RENTAL_PRICE, KEY_RENTAL_PRICE);
    map.put(KEY_RATING_STYLE, KEY_RATING_STYLE);
    map.put(KEY_RATING_SCORE, KEY_RATING_SCORE);
    map.put(KEY_PRODUCTION_YEAR, KEY_PRODUCTION_YEAR);
    map.put(KEY_COLUMN_DURATION, KEY_COLUMN_DURATION);
    map.put(KEY_ACTION, KEY_ACTION);
    map.put(BaseColumns._ID, "rowid AS " +
            BaseColumns._ID);
    map.put(SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID, "rowid AS " +
            SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID);
    map.put(SearchManager.SUGGEST_COLUMN_SHORTCUT_ID, "rowid AS " +
            SearchManager.SUGGEST_COLUMN_SHORTCUT_ID);
    return map;
  }
...
```

在上面的例子中，注意填充SUGGEST_COLUMN_INTENT_DATA_ID字段。这是URI的一部分，指向独一无二的内容到这一列的数据，那是URI描述的内容被存储的最后部分。在URI的第一部分，与所有表格的列同样，是设置在searchable.xml文件，用android:searchSuggestIntentData属性。属性被描述在Handle Search Suggestions。

如果URI的第一部分是不同于表格的每一列，我们填充SUGGEST_COLUMN_INTENT_DATA字段的值。当用户选择这个内容时，这个intent被启动依据SUGGEST_COLUMN_INTENT_DATA_ID的混合intent数据或者 `android:searchSuggestIntentData` 属性和SUGGEST_COLUMN_INTENT_DATA字段值之一。

提供搜索建议数据

实现一个Content Provider去返回搜索术语建议到AndroidTV搜索框。系统需要我们的内容容器提供建议，通过调用每次一个字母类型query())方法。在query())的实现中，我们的内容容器搜索我们的建议数据并且返回一个光标指向我们已经指定的建议列。

```
@Override
  public Cursor query(Uri uri, String[] projection, String selection, String[] selecti
onArgs,
                      String sortOrder) {
    // Use the UriMatcher to see what kind of query we have and format the db query ac
cordingly
    switch (URI_MATCHER.match(uri)) {
      case SEARCH_SUGGEST:
          Log.d(TAG, "search suggest: " + selectionArgs[0] + " URI: " + uri);
          if (selectionArgs == null) {
              throw new IllegalArgumentException(
                      "selectionArgs must be provided for the Uri: " + uri);
          }
          return getSuggestions(selectionArgs[0]);
      default:
          throw new IllegalArgumentException("Unknown Uri: " + uri);
    }
  }

  private Cursor getSuggestions(String query) {
    query = query.toLowerCase();
    String[] columns = new String[]{
      BaseColumns._ID,
      VideoDatabase.KEY_NAME,
      VideoDatabase.KEY_DESCRIPTION,
      VideoDatabase.KEY_ICON,
      VideoDatabase.KEY_DATA_TYPE,
      VideoDatabase.KEY_IS_LIVE,
      VideoDatabase.KEY_VIDEO_WIDTH,
      VideoDatabase.KEY_VIDEO_HEIGHT,
      VideoDatabase.KEY_AUDIO_CHANNEL_CONFIG,
      VideoDatabase.KEY_PURCHASE_PRICE,
      VideoDatabase.KEY_RENTAL_PRICE,
      VideoDatabase.KEY_RATING_STYLE,
      VideoDatabase.KEY_RATING_SCORE,
      VideoDatabase.KEY_PRODUCTION_YEAR,
      VideoDatabase.KEY_COLUMN_DURATION,
      VideoDatabase.KEY_ACTION,
      SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID
    };
    return mVideoDatabase.getWordMatch(query, columns);
  }
...
```

在我们的manifest文件中，内容容器接受特殊处理。相比被标记为一个activity，它是被描述为[provider](http://developer.android.com/guide/topics/manifest/provider-element.html)。provider包括 `android:searchSuggestAuthority` 属性去告诉系统我们的内容容器的名字空间。并且，我们必须设置它的 `android:exported` 属性为 `"true"` ，这样Android全局搜索能用它返回的搜索结果。

```
<provider android:name="com.example.android.tvleanback.VideoContentProvider"
    android:authorities="com.example.android.tvleanback"
    android:exported="true" />
```

处理搜索建议

我们的应用必须包括res/xml/searchable.xml文件去配置搜索建议设置。它包括android:searchSuggestAuthority属性去告诉系统内容容器的名字空间。这必须匹配在 `AndroidManifest.xml` 文件的[provider](http://developer.android.com/guide/topics/manifest/provider-element.html)元素的android:authorities 属性的字符串值。

searchable.xml文件必须也包含在 `"android.intent.action.VIEW"` 的android:searchSuggestIntentAction值去定义提供自定义建议的intent action。这与提供一个搜索术语的intent action不同，下面解释。查看Declaring the intent action 用另一种方式去定义建议的intent action。

同intent action一起，我们的应用必须提供我们定义的android:searchSuggestIntentData属性的intent数据。这是指向内容的URI的第一部分。它描述在填充的内容表格中URI所有共同列的部分。URI的独一无二的部分用 SUGGEST_COLUMN_INTENT_DATA_ID字段建立每一列，以上被描述在识别列。查看Declaring the intent data用另一种方式去定义建议的intent数据。

并且，注意 `android:searchSuggestSelection="?"` 属性为特定的值。这个值作为query())方法 `selection` 参数。方法的问题标记(?)值被代替为请求文本。

最后，我们也必须包含android:includeInGlobalSearch属性值为 `"true"` 。这是一个searchable.xml文件的例子：

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
        android:hint="@string/search_hint"
        android:searchSettingsDescription="@string/settings_description"
        android:searchSuggestAuthority="com.example.android.tvleanback"
        android:searchSuggestIntentAction="android.intent.action.VIEW"
        android:searchSuggestIntentData="content://com.example.android.tvleanback/vide
o_database_leanback"
        android:searchSuggestSelection=" ?"
        android:searchSuggestThreshold="1"
        android:includeInGlobalSearch="true"
    >
</searchable>
```

处理搜索术语

一旦搜索框有一个字匹配到了应用列中的一个（被描述在上文的识别列），系统启动ACTION_SEARCH intent。我们应用的activity处理intent搜索列的给定的字段资源，并且返回一个那些内容项的列表。在我们的 `AndroidManifest.xml` 文件中，我们指定的activity处理ACTION_SEARCH intent，像这样：

```
...
  <activity
      android:name="com.example.android.tvleanback.DetailsActivity"
      android:exported="true">

      <!-- Receives the search request. -->
      <intent-filter>
          <action android:name="android.intent.action.SEARCH" />
          <!-- No category needed, because the Intent will specify this class componen
t -->
      </intent-filter>

      <!-- Points to searchable meta data. -->
      <meta-data android:name="android.app.searchable"
          android:resource="@xml/searchable" />
  </activity>
...
  <!-- Provides search suggestions for keywords against video meta data. -->
  <provider android:name="com.example.android.tvleanback.VideoContentProvider"
      android:authorities="com.example.android.tvleanback"
      android:exported="true" />
...
```

activity必须参考searchable.xml文件描述可搜索的设置。用全局搜索框，manifest必须描述activity应该收到的搜索请求。manifest必须描述[provider](http://developer.android.com/guide/topics/manifest/provider-element.html)元素，详细被描述在searchable.xml文件。

深链接到应用的详情页

如果我们有设置处理搜索建议描述的搜索配置和填充SUGGEST_COLUMN_TEXT_1，SUGGEST_COLUMN_CONTENT_TYPE和SUGGEST_COLUMN_PRODUCTION_YEAR字段到识别列，一个深链接去查看详情页的内容。当用户选择一个搜索结果时，详情页将打开。如图1。

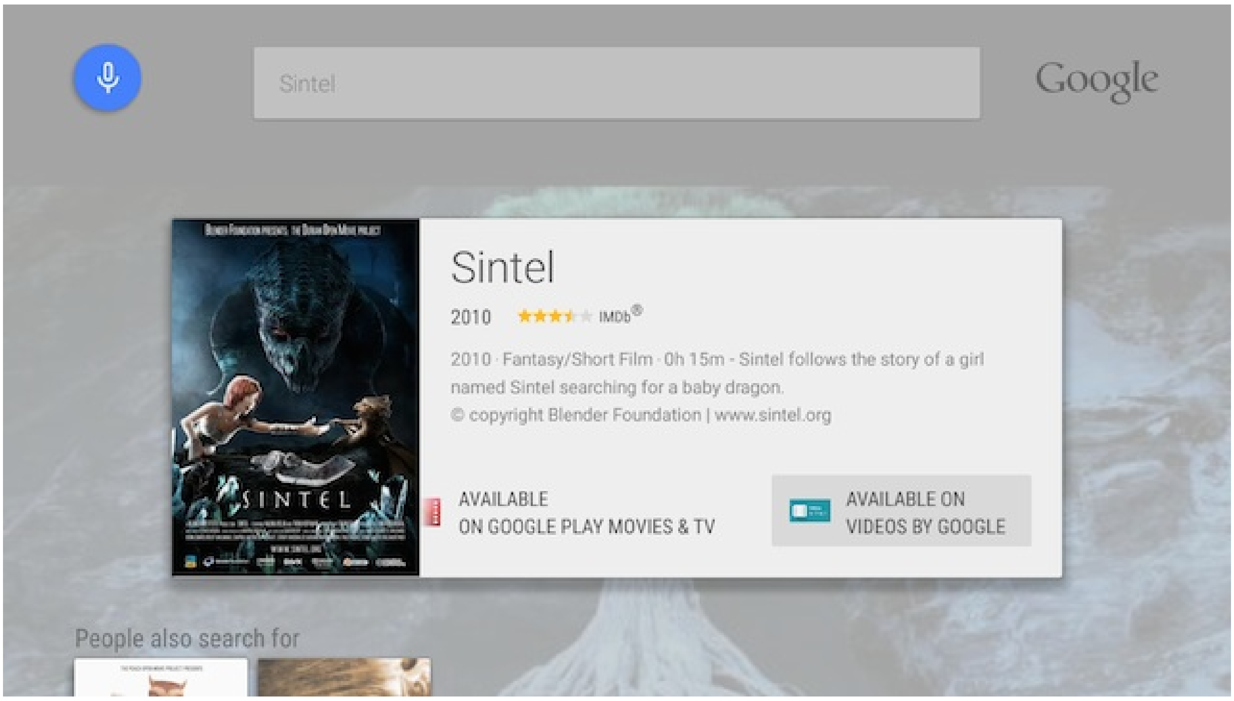


图**1** 详情页显示一个深链接为Google(Leanback)的视频代码。Sintel: © copyright Blender Foundation, www.sintel.org.

当用户选择我们的应用链接， `“Available On”` 按钮被标识在详情页，系统启动activity处理ACTION_VIEW（在searchable.xml文件设置android:searchSuggestIntentAction值为 `"android.intent.action.VIEW"` ）。

我们也能设置用户intent去启动我们的activity，这个在在AndroidLeanback示例代码应用中演示。注意示例应用启动它自己的 `LeanbackDetailsFragment` 去显示被选择媒体的详情，但是我们应该启动activity去播放媒体。立即去保存用户的另一次或两次点击。

下一节: 使TV应用是可被搜索的 >

TV应用内搜索

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/discovery/in-app-search.html

当在TV上用媒体应用时，用户脑中通常有期望的内容。如果我们的应用包含一个大的内容目录，为用户找到他们想找到的内容时，用特定的标题浏览可能不是最有效的方式。一个搜索界面能帮助用户获得他们想快速浏览的内容。

Leanback support library提供一套类库去使用标准的搜索界面。在我们的应用内使用类库，可以和TV其他搜索功能，如语音搜索，获得一致性。

这节课讨论如何在我们的应用中用Leanback支持类库提供搜索界面。

添加搜索操作

当我们用BroweseFragment类做一个媒体浏览界面时，我们能使用搜索界面作为用户界面的一个标准部分。当我们设置View.OnClickListener在BrowseFragment对象时，搜索界面作为一个图标出现在布局中。接下来的示例代码展示了这个技术。

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.browse_activity);

    mBrowseFragment = (BrowseFragment)
            getFragmentManager().findFragmentById(R.id.browse_fragment);

    ...

    mBrowseFragment.setOnSearchClickedListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(BrowseActivity.this, SearchActivity.class);
            startActivity(intent);
        }
    });

    mBrowseFragment.setAdapter(buildAdapter());
}
```

Note: You can set the color of the search icon using the setSearchAffordanceColor(int).-->

> **Note**：我们能设置搜索图标的颜色用setSearchAffordanceColor(int))。

添加搜索输入和结果展示

当用户选择搜索图标，系统通过定义的intent关联一个搜索activity。我们的搜索activity应该用包括SearchFragment的线性布局。这个fragment必须实现SearchFragment.SearchResultProvider界面去显示搜索结果。

接下来的示例代码展示了如何扩展SearchFragment类去提供搜索界面和结果：

```
public class MySearchFragment extends SearchFragment
        implements SearchFragment.SearchResultProvider {

    private static final int SEARCH_DELAY_MS = 300;
    private ArrayObjectAdapter mRowsAdapter;
    private Handler mHandler = new Handler();
    private SearchRunnable mDelayedLoad;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mRowsAdapter = new ArrayObjectAdapter(new ListRowPresenter());
        setSearchResultProvider(this);
        setOnItemClickedListener(getDefaultItemClickedListener());
        mDelayedLoad = new SearchRunnable();
    }

    @Override
    public ObjectAdapter getResultsAdapter() {
        return mRowsAdapter;
    }

    @Override
    public boolean onQueryTextChange(String newQuery) {
        mRowsAdapter.clear();
        if (!TextUtils.isEmpty(newQuery)) {
            mDelayedLoad.setSearchQuery(newQuery);
            mHandler.removeCallbacks(mDelayedLoad);
            mHandler.postDelayed(mDelayedLoad, SEARCH_DELAY_MS);
        }
        return true;
    }

    @Override
    public boolean onQueryTextSubmit(String query) {
        mRowsAdapter.clear();
        if (!TextUtils.isEmpty(query)) {
            mDelayedLoad.setSearchQuery(query);
            mHandler.removeCallbacks(mDelayedLoad);
            mHandler.postDelayed(mDelayedLoad, SEARCH_DELAY_MS);
        }
        return true;
    }
}
```

上面的示例代码展示了在分开的线程用独立的 `SearchRunnable` 类去运行搜索请求。这个技巧是从正在阻塞的主线程保持了潜在的慢运行请求。

下一节: 创建TV游戏应用 >

创建TV游戏应用

> 编写:dupengwei - 原文:http://developer.android.com/training/tv/games/index.html

TV屏幕为手机游戏开发者提供了大量的新思考。这些领域包括它的大尺寸，它的控制方案和所有玩家可以同时观看的事实。

显示器

开发TV游戏时有两点要记住，就是TV屏幕具有共享显示器的特性，和横向设计游戏的需求。

考虑共享显示

客厅TV带来了多人游戏的设计挑战，客厅TV游戏时所有玩家都可以看到。这个问题与游戏，特别是依靠每个玩家用于隐藏信息的游戏（如纸牌游戏、战略游戏）息息相关。 我们可以通过实现一些机制来解决一个玩家窃取另一玩家信息的问题。这些机制是：

- 屏幕罩可以帮助隐藏信息。例如，在一个回合制游戏，像单词或卡片游戏，一次只有一个玩家能看到显示的内容。当这个玩家完成一个步骤，游戏允许他用一个能阻碍其他人看到秘密信息的罩遮住屏幕。当下一个玩家开始操作，这个罩就会打开显示他自己的信息。
- 在手机或平板电脑上运行一个伙伴app作为第二屏幕，通过这种方式让玩家隐藏信息。

支持横向显示

TV总是单向显示的：我们不能翻转它的屏幕，且没有纵向显示。要总是以横向显示模式设计我们的TV游戏。

输入设备

TV没有触摸屏接口，所以更重要的是获取控制要正确，并确保玩家使用起来要直观和有趣。处理控制器还介绍了其他一些问题需要注意，如跟踪多个控制器，，处理断开要适当。

支持**D-pad**控制

围绕方向键（D-pad）控制来计划我们的控制方案，因为这种控制是Android TV设备的默认设置。玩家需要在游戏的所有方面使用方向键（D-pad）——不仅仅是控制核心游戏设置，而且能导航菜单和广告。因此，我们还应该确保我们的Android TV游戏不能涉及触摸屏。例如，

一个Android TV游戏不应该告诉玩家> 点击这里继续。 如何塑造玩家使用控制器与游戏进行互动的方式将是实现良好用户体验的关键：

- 通信控制器的要求。利用Android市场上app的产品描述将控制器的期望传达给玩家们。如果一个游戏使用摇杆游戏手柄比只用一个方向键更合适，请将这一事实说清楚。玩家使用一个不适合游戏的控制器玩游戏很可能导致游戏体验欠佳，从而对游戏的评价造成不利影响。
- 使用一致的按钮映射。直观和灵活的按钮映射是良好用户体验的关键。例如，我们应该遵守使用A按钮接受，而B按钮取消的既定习惯。我们也可以提供重映射形式方面的灵活性。关于按钮映射的更多信息，参见Handling Controller Actions。
- 检测控制器功能并相应地调整。查询控制器的能力以优化控制器和游戏直接的匹配程度。例如，我们可能打算让一个玩家通过摇晃控制器来控制一个对象。然而，如果玩家的控制器缺少加速计和陀螺仪硬件设施，摇晃控制器并不会产生效果。所以，我们的游戏应该检查控制器，如果该控制器不支持运行检测，则切换到另一个可用的控制方案。更多关于检测控制器功能的信息，参见Controllers Across Android Versions。

提供适当的后退按钮的行为

返回按钮不应该作为切换。例如，不能使用它打开和关闭一个菜单。它应该只能导航后退，breadcrumb-style，玩家之前访问过屏幕页面，例如：游戏界面>游戏暂停界面>游戏主界面>Android主界面。 由于返回按钮应该只能进行线性导航（后退），我们可以使用返回按钮离开一个游戏内菜单（由不同的按钮打开），回到游戏界面。更多关于导航设计的信息，参见Navigation with Back and Up。学习更多关于实现的信息，参见Providing Proper Back Navigation。

使用适当的按钮

并不是所有的游戏控制器提供开始,搜索,或菜单按钮。确保我们的UI不取决于这些按钮的使用。

处理多个控制器

当多个玩家玩游戏,每个都有他或她自己的控制器，做好每对“玩家-控制器”的映射是很重要的。关于如何实现“控制器-数量”识别的信息，参见Input Devices。

处理控制器的断开

当控制器从游戏中断开时，游戏应该暂停，并弹出一个对话框促使断开的玩家重新连接他或她的控制器。 对话框还应提供排除故障的提示（如，一个弹出的对话框告诉玩家“检查我们的蓝牙连接”）关于实现输入设备支持的更多信息,参见Handling Controller Actions。具体关于蓝牙连接的信息，参见Bluetooth。

展示控制器说明

如果我们的游戏提供了可视化的游戏控制说明，控制器图片应该是免费的、品牌化的，并且只能包含与Android兼容的按钮。 Android兼容的控制器样图，点击Android TV Gamepad Template (ZIP)下载。它包含一个黑底的白色控制器和一个白底的黑色控制器，是一个PNG类型的Adobe®Illustrator®文件。

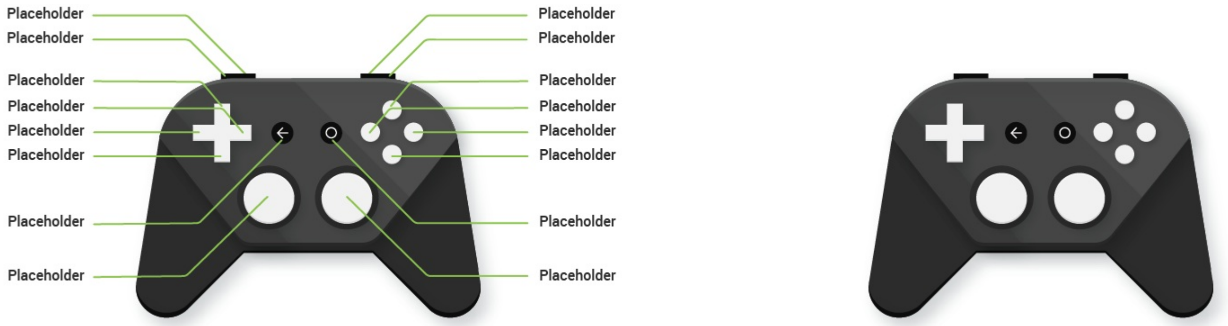


Figure 1. 控制器说明的示例请使用Android TV Gamepad Template (ZIP)

Manifest

有一些特殊的东西应该包含在游戏的Android Manifest里。

在屏幕主界面显示游戏

Android TV主界面采用单独一行来显示游戏，与常规应用分开显示。为了让游戏出现在游戏列表，设置游戏的manifest清单的标签下的 `android:isGame` 属性为 `"true"` 。例如：

```
<application
    ...
    android:isGame="true"
    ...
>
```

声明游戏控制器支持

游戏控制器对于TV设备的用户来说可能不是有效的。为了适当的通知用户，游戏需要（或只支持）一个控制器，我们必须在app的manifest里包含这些条目。如果我们需要一个游戏控制器，我们必须在app的manifest中包含以下条目：

```
<uses-feature android:name="android.hardware.gamepad"/>
```

如果我们的游戏使用了一个游戏控制器，但是不需要，在app的manifest里包含以下的功能条目：

```
<uses-feature android:name="android.hardware.gamepad" android:required="false"/>
```

更多关于manifest条目的信息，参见App Manifest。

Google Play Game 服务

如果我们的游戏集成了Google Play Game 服务，我们应该记住一些关于成果的注意事项，登录，保存游戏，和多人游戏。

成就

我们的游戏应包含至少5个(可获取的)成果。只有一个用户从一个受支持的输入设备控制游戏应该能够获得成就。关于成就的更多信息以及如何实现，参见Achievements in Android。

登录

我们的游戏应该试图在启动的时候让用户登录。如果玩家连续几次拒绝登录后，游戏应该停止询问。学习更多关于登录的信息在Implementing Sign-in on Android。

保存

使用Google Play Services保存游戏来存储保存的游戏。我们应该讲保存的游戏绑定到一个特定的谷歌账号，作为唯一标识，甚至在跨设备时也不受影响。无论玩家使用手机或TV，游戏应该可以从同一个用户账号获取到保存的游戏信息。

我们也应该在我们的游戏的UI提供一个选项,让玩家删除本地和云存储端的数据。我们可能把选项放在游戏的设置界面。使用Play Services保存游戏的实现细节，参见Saved Games in Android

多人游戏

一个游戏要提供多人游戏体验，必须允许至少2个玩家进入一个房间。进一步了解Android的多人游戏信息，参见Android developer网站的Real-time Multiplayer和Turn-based Multiplayer文档。

退出

提供一个一致和明显的UI元素,让用户适当的退出游戏。这个元素应该用方向键导航按钮访问，这样做而不是依赖Home键提供退出功能，是因为在使用不同的控制器时，若依赖Home键提供退出功能，这既不一致也不可靠。

Web

不要让android TV的游戏浏览网页。Android TV不支持web浏览器。

> **Note**：我们可以使用WebView类实现登录像Google+ 和 Facebook这样的服务。

网络

游戏经常需要更大的带宽提供最佳的性能,许多用户宁愿选择有线网而不愿选择WiFi来提供性能。我们的app应该对有线网和WiFi连接都进行检查。如果我们的app只针对TV，我们不需要检查3G/LTE服务，而移动app则需要检查3G/LTE服务。

下一节: TV应用清单 >

创建TV直播应用

> 编写:dupengwei - 原文:http://developer.android.com/training/tv/tif/index.html

看电视直播节目和其他连续的、基于频道的内容是TV体验的主要部分。Android 通过Android 5.0中的TV Input Framework支持直播视频内容的接收和重放（API Level 21）。该框架提供了一个统一的方法，从硬件源（如HDMI端口和内置调谐器）和软件源（如流传在互联网上的视频）接收音频和视频内容。

该框架能使开发人员通过实现TV输入服务定义直播TV输入源。该服务发布一个频道和节目列表到一个TV Provider上。电视设备的直播电视应用从TV Provider获取可用的频道和节目列表并显示给用户。当用户选择某个特定的频道，直播TV应用软件通过TV Input Manager为相关TV输入服务创建一个会话，并告诉TV输入服务调整到请求频道，然后将内容显示到TV应用软件提供的显示器上。

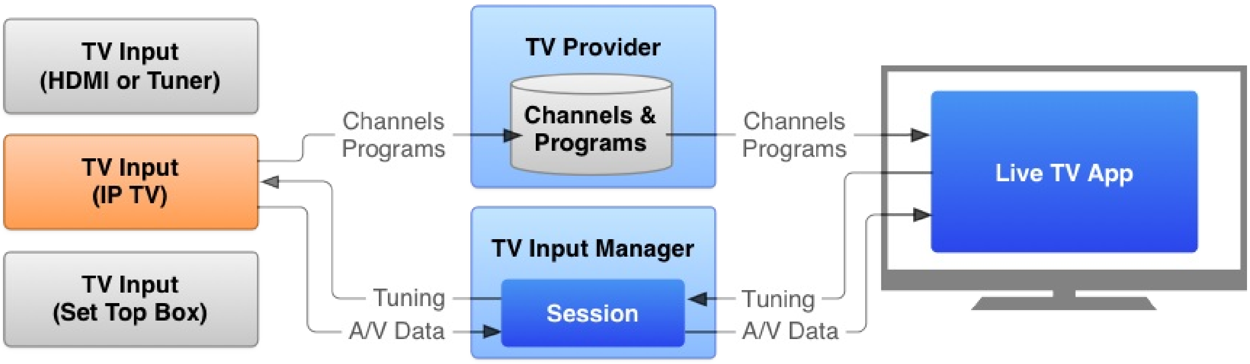


图1.电视输入框架功能图

TV Input Framework 的设计目的是提供各种各样的TV输入源并把它们整合到一个单一的用户界面供用户浏览、查看和享受内容。为你想要传播的节目构建一个TV输入服务之后，用户可以更加轻易地通过TV设备收看这些节目。

更多关于TV输入框架的信息，请参考android.media.tv。

TV应用清单

> 编写:awong1900 - 原文:http://developer.android.com/training/tv/publishing/checklist.html

用户喜欢的TV应用应是体验一致的，有逻辑的和可预测的。他们可以在应用内四处浏览，并且不会迷失在应用从而重设UI导致重头开始。用户欣赏干净的，有色彩的和起作用的界面，这样的体验会很好。把这些想法放在脑子中，我们能创造适合Android TV的应用并达到用户的期望。

这个清单覆盖了应用和游戏的开发的主要方面去确保我们的应用提供了最好的体验。额外的游戏注意事项仅被包含在游戏小节。

关于Google Play中Android TV应用的质量标准，参考TV App Quality。

TV格式因素的支持

这些清单项目使用在游戏和应用中。

1. 确定manifest的主activity使用 `CATEGORY_LEANBACK_LAUNCHER` 。 查看Declare a TV Activity。
2. 提供每种语言的主屏幕横幅支持。
    - 启动应用横幅大小为320x180 px
    - 横幅资源放在 `drawables/xhdpi` 目录
    - 横幅图像包含本地化的文本去识别应用。 查看Provide a home screen banner。
3. 消除不支持的硬件要求。 查看Declaring hardware requirements for TV。
4. 确保没有隐式的权限需求。 查看Declaring permissions that imply hardware features。

用户界面设计

这些清单项使用在游戏和应用中。

1. 提供适合横屏模式的布局资源。 查看 Build Basic TV Layouts。
2. 确保文本和控件在一定距离外看是足够大的。 查看Build Useable Text and Controls。
3. 为HDTV屏幕提供高分辨率的位图和图标。 查看 Manage Layout Resources for TV。
4. 确保我们的图标和logo符合Android TV的规范。 查看Manage Layout Resources for TV。
5. 允许布局使用overscan。 查看Overscan。
6. 使每一个布局元素都能用D-pad和游戏控制器操作。 查看 Creating Navigation 和

Handling Controllers。

7. 当用户通过文本搜索时改变背景图像。 查看Update the Background。
8. 在Leanback fragments中定制背景颜色去匹配品牌。 查看Customize the Card View。
9. 确保我们的UI不需要触摸屏。 查看Touch screen and Declare touch screen not required。
10. 遵循有效的广告的指导。 查看Provide Effective Advertising。

搜索和发现内容

这些清单项使用在游戏和应用中。

1. 在Android TV全局搜索框中提供搜索结果。 查看Provide Data。
2. 提供TV特定数据字段的搜索。 查看Identify Columns。
3. 确保应用的详情屏幕有可发现的内容以便用户立即开始观看。 查看Display Your App in the Details Screen。
4. 放置相关的，可操作的内容和目录在主屏幕，使用户容易的发现内容。 查看Recommending TV Content。

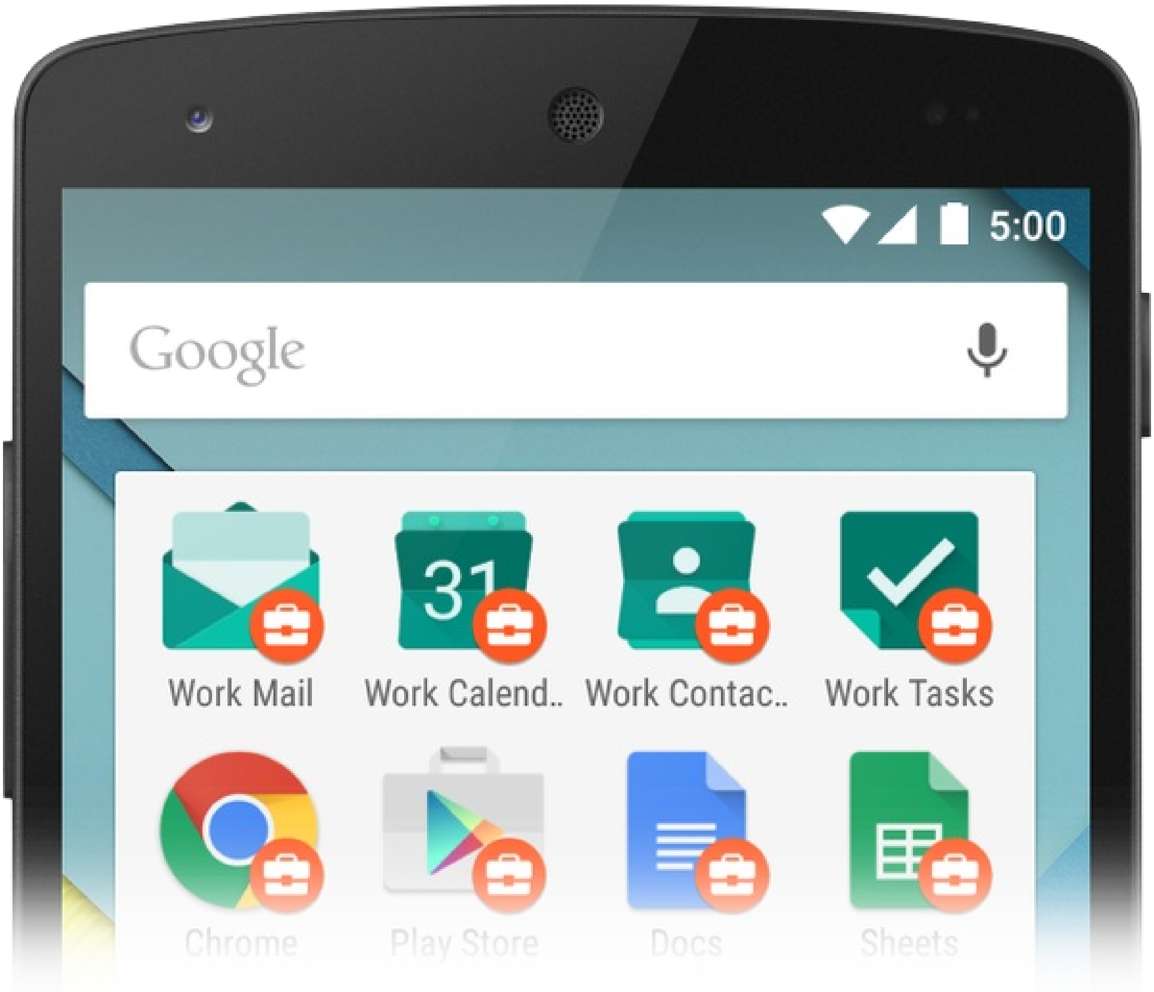
游戏

这些清单项目使用在游戏。

1. 在manifest中用 `isGame` 标记让游戏显示在主屏幕上。 查看Show your game on the home screen。
2. 确保游戏控制器可以不依靠开始，选择，或者菜单键操作(不是所有控制器有这些按键)。 查看Input Devices。
3. 使用通常的游戏手柄布局（不包括特殊的控制器品牌）去显示游戏按键示意图。 查看Show controller instructions。
4. 检查网络和WiFi连接。 查看Networking。
5. 提供给用户清晰的退出提示。 查看Exit。

创建企业级应用

> 编写:craftsmanBai - http://z1ng.net - 原文:http://developer.android.com/training/enterprise/index.html



Android框架提供安全支持、数据分离、企业环境管理的功能。作为应用开发者，通过适当地处理企业安全和功能限制，你可以让你的应用程序吸引更多的企业客户。也可以修改你的应用使技术管理员可远程配置使用企业资源。

为了帮助企业将安卓设备和应用程序进入工作场所，Google通过Android for Work为设备的分配和管理提供了一套API和服务。通过这项计划，企业可以连接到企业移动性管理（EMM）供应商，将Android整合到工作中。

通过下面的链接获取，可以了解更多关于如何更新您的Android应用程序来支持企业环境或建立企业解决方案的信息。

企业级应用开发

了解在企业环境中如何使您的应用程序运行顺畅，限制设备的功能和数据访问。通过加入限制进一步支持企业使用你的app，让管理员可以远程配置使用你的应用程序：

确保与管理兼容：

http://developer.android.com/training/enterprise/app-compatibility.html

加入应用限制：

http://developer.android.com/training/enterprise/app-restrictions.html

应用限制计划：

http://developer.android.com/samples/AppRestrictionSchema/index.html

应用限制执行者：

http://developer.android.com/samples/AppRestrictionEnforcer/index.html

设备与应用管理

学习如何为应用程序建立策略控制器，使企业的技术管理人员来管理设备，管理企业应用程序，并提供访问公司资源的权限：

建立工作策略控制：

http://developer.android.com/training/enterprise/work-policy-ctrl.html

基本管理模型：

http://developer.android.com/samples/BasicManagedProfile/index.html

利用 Managed Profile 确保兼容性

> 编写：zenlynn 原文：http://developer.android.com/training/enterprise/app-compatibility.html

Android 平台允许设备有 managed profile。managed profile 由管理员控制，它的功能和用户原本的 profile 的功能是分别设置的。通过这种方法，在用户设备上运行的企业所定制应用程序和数据的环境就在企业的控制之下，同时用户还能使用私人的应用程序和 profile。

本节课展示了如何修改你的应用程序，使之能够在有 managed profile 的设备上可靠运行。除了一般应用开发的最佳实践外，你不用做任何事。然而，在有 managed profile 的设备上，最佳实践的其中一些规范变得尤为重要。本文件强调了你所需要了解的问题。

概述

用户经常想在企业环境中使用他们的私人设备。这种情况可能让企业陷入困境。如果用户使用他们的私人设备，企业不得不担心在这个不受控制的设备上的机密信息（例如员工的电子邮件和通讯录）。

为了处理这种情况，Android 5.0（API 21）允许企业设置 managed profile。如果设备有 managed profile，这个 profile 的设置是在企业管理员的控制之下的。管理员可以选择在这个 profile 之下，什么应用程序可以运行，什么设备功能可以允许。

如果一个设备有 managed profile，那么，无论应用程序在哪个 profile 之下运行，都意味着：

- 默认情况下，大部分的 intent 无法从一个 profile 跨越到另一个。如果在某个 profile 之下的一个应用程序创建了 intent，而这个 profile 无法响应，又因为 profile 的限制这个 intent 不允许跨越到其他 profile，那么，这个请求就失败了，应用程序可能意外关闭。
- profile 管理员可以在 managed profile 中限制哪个系统应用程序可以运行。这个限制可能导致在 managed profile 中一些常见的 intent 无法处理。
- 因为 managed profile 和非 managed profile 有各自的存储区域，导致文件 URI 在一个 profile 中有效，但在其他 profile 中无效。在一个 profile 中创建的 intent 可能在其他 profile（取决于 profile 设置）中被响应，所以在 intent 中放置文件 URI 是不安全的。

防止失败的 intent

在一个有 managed profile 的设备上，intent 是否能从一个 profile 跨越到另一个，存在着限制。大多情况下，一个 intent 在哪个 profile 中创建，就在哪个 profile 中响应。如果那个 profile 中无法响应，就算在其他 profile 中可以响应，这个 intent 也不会被响应，而且创建这个 intent 的应用程序会意外关闭。

profile 管理员可以选择哪个 intent 可以从一个 profile 跨越到另一个。因为是由管理员做决定，所以你无法预先知道哪个 intent 可以跨越边界。管理员设置了这个策略，而且可以在任何时候自由更改。

在你的应用程序启动一个 activity 之前，你应该验证这是可行的。你可以调用 Intent.resolveActivity() 方法来验证。如果无法处理，方法会返回 null。如果方法返回值非空，那么至少有一个方法可以处理这个 intent，所以创建这个 intent 是安全的。这种情况下，或者是因为在当前 profile 中可以响应，或者是因为 intent 被允许跨越到可以处理的其他 profile 中，intent 可以被处理。（更多关于响应 intent 的信息，请查看 Common Intents。）

例如，如果你的应用程序需要设置定时器，就需要检查是否能响应 ACTION_SET_TIMER intent。如果应用程序无法响应这个 intent，就需要采取恰当的行动（例如显示一个错误信息）。

```
public void startTimer(String message, int seconds) {

    // Build the "set timer" intent
    Intent timerIntent = new Intent(AlarmClock.ACTION_SET_TIMER)
            .putExtra(AlarmClock.EXTRA_MESSAGE, message)
            .putExtra(AlarmClock.EXTRA_LENGTH, seconds)
            .putExtra(AlarmClock.EXTRA_SKIP_UI, true);

    // Check if there's a handler for the intent
    if (timerIntent.resolveActivity(getPackageManager()) == null) {

        // Can't resolve the intent! Fail this operation cleanly
        // (perhaps by showing an error message)

    } else {
        // Intent resolves, it's safe to fire it off
        startActivity(timerIntent);

    }
}
```

跨越 profile 共享文件

有时候应用程序需要授权其他运用程序访问自己的文件。例如，一个图片库应用可能想与图片编辑器共享它的图片。一般共享文件有两种方法：通过文件 URI 或者内容 URI。

一个文件 URI 是由前缀 `file:` 和文件在设备中存储的绝对路径组成的。然而，因为 managed profile 和私人 profile 有各自的存储区域，所以一个文件 URI 在一个 profile 中是有效的，在其他 profile 中是无效的。这种情况意味着，如果你要在 intent 中放置一个文件 URI，而这个 intent 要在其他 profile 中响应，那么响应方是不能访问这个文件的。

你应该取而代之用内容 URI 共享文件。内容 URI 用一种更安全、更易于分享的方式来识别文件。内容 URI 包括了文件路径，文件提供者，以及文件 ID。你可以通过 FileProvider 为任何文件生成内容 ID。然后，你就可以和（甚至在其他 profile 中的）其他应用程序共享内容 ID。响应方可以使用内容 ID 来访问实际文件。

例如，这里展示了你怎么获得一个指定文件 URI 的内容 URI：

```
// Open File object from its file URI
File fileToShare = new File(fileUriToShare);

Uri contentUriToShare = FileProvider.getUriForFile(getContext(),
        "com.example.myapp.fileprovider", fileToShare);
```

当你调用 getUriForFile() 方法时，必须包括文件提供者的权限（在这个例子里是 `"com.example.myapp.fileprovider"` ），在应用程序的 manifest 中，用 \ 元素设定这个权限。更多关于用内容 URI 共享文件的信息，请查看共享文件。

在 managed profile 环境测试你的应用程序的兼容性

你要在有 managed profile 的环境中测试你的应用程序，以发现会引起运行失败的问题。在一个有 managed profile 的设备中测试是一个验证你的应用程序正确响应 intent 的好办法：无法响应的时候不创建 intent，不使用无法跨越 profile 的 URI 等等。

我们提供了一个示例应用程序，BasicManagedProfile，你可以用它在一个运行 Android 5.0 或者更高系统的 Android 设备上设置一个 managed profile。这个应用程序为在有 managed profile 的环境中来测试你的应用程序提供了一个简单的方法。你也可以按照下面的方法用这个应用程序来设置你的 managed profile：

- 在 managed profile 中设定哪些默认应用程序可以使用
- 设定哪些 intent 被允许从一个 profile 跨越到另一个

如果你通过 USB 线手动安装一个有 managed profile 的应用程序，那么在 managed profile 和非 managed profile 之中都安装有这个应用程序。只要你安装了应用程序，你就能在以下条件下进行测试：

- 如果一个 intent 可以被一个默认的应用程序（例如相机应用程序）响应，试试 managed

profile 中禁用这个默认应用程序，然后验证这个应用程序可以做出恰当的行为。

- 如果你创建了一个 intent 希望被其他应用程序响应，试试启用以及禁用这个 intent 从一个 profile 跨越到另一个的权限。验证在这两种情况下应用程序都能做出恰当的行为。如果 intent 不允许在 profile 之间跨越，无论当前 profile 是否能做出响应，都要验证应用程序能做出恰当的行为。例如，如果你的应用程序创建了一个地图相关的 intent，试试以下每一种情况：
    - 设备允许地图 intent 从一个 profile 跨越到另一个，并且在另一个（并非应用程序所运行的）profile 之中有恰当的响应
    - 设备不允许地图 intent 在 profile 之间跨越，但是在应该程序所运行的 profile 之中有恰当的响应
    - 设备不允许地图 intent 在 profile 之间跨越，并且在设备的 profile 之中没有恰当的响应

- 如果你在 intent 里放置了内容，不管是在当前 profile 之中，还是在跨越 profile 之后，都要验证 intent 能有恰当的行为。

在 managed profile 环境测试：提示与技巧

你会发现在有 managed profile 的设备里进行测试有一些技巧。

- 如前所述，当你侧载一个应用程序到一个有 managed profile 的设备里，是在 managed profile 和非 managed profile 之中都安装了。如果你愿意，你可以从一个 profile 之中删除，在另一个 profile 之中留下。

- 在安卓调试桥（adb）shell 端可用的 activity manager 命令大部分都支持 `--user` 标识，你可以用之设定运行应用程序的用户。通过设定一个用户，你可以选择是在 managed profile 之中运行，还是在非 managed profile 之中运行。更多信息，请查看 ADB Shell Commands。

- 为了找到设备上的活跃用户，使用 adb 包管理器的 `list users` 命令。输出的字符串中第一个数字是用户 ID，你可以用于 `--user` 标识。更多信息，请查看 ADB Shell Commands。

例如，为了找到一个设备上的用户，你会运行这个命令：

```
$ adb shell pm list users
UserInfo{0:Drew:13} running
UserInfo{10:Work profile:30} running
```

在这里，非 managed profile（"Drew"）有个 ID 为 0 的用户，而 managed profile 有个 ID 为 10 的用户。要在工作 profile 之中运行一个应用程序，你会用到这样的命令：

```
$ adb shell am start --user 10 \
-n "com.example.myapp/com.example.myapp.testactivity" \
-a android.intent.action.MAIN -c android.intent.category.LAUNCHER
```

实现 app 的限制

> 编写：zenlynn 原文：http://developer.android.com/training/enterprise/app-restrictions.html

如果你为企业市场开发 app ，你可能需要满足企业政策的特殊要求。应用程序的限制允许企业管理员远程设定 app 。这种能力对于部署了 managed profile 的企业 app 来说，尤其有用。

例如，一个企业可能需要核准的 app 允许企业管理员：

- 为一个网页浏览器添加白名单或黑名单网址
- 配置是否允许一个 app 通过蜂窝网络同步内容，或只能通过 Wi-Fi
- 配置 app 的电子邮件设定

这个指南展示了如何在你的 app 实现这个配置设定。

> 注意：由于历史原因，这些配置设定被称为限制，并在文件与类中使用这个术语（例如 RestrictionsManager）。然而，这些限制实际上可以实现各种各样的配置选项，并不只是限制 app 的功能。

远程配置概述

app 定义了管理员可以远程设定的限制和配置选项。限制提供者可以随意改变配置设定。如果你的 app 运行在企业设备上的 managed profile 中 ，企业管理员可以改变该 app 的限制。

限制提供者是运行在同一个设备上的另一个 app 。这个 app 通常是由企业管理员控制。企业管理员向限制提供者 app 传达限制的改变。这个 app 就相应地改变你的 app 的限制。

提供外部可配置的限制：

- 在你 app 的 manifest 中声明限制。这么做允许企业管理员通过 Goodle Play 的接口读取 app 的限制。
- 每当 app 恢复，使用 RestrictionsManager 对象检查当前限制，并改变你的 app 的 UI 和行为以符合这些限制。
- 监听 ACTION_APPLICATION_RESTRICTIONS_CHANGED intent。当你收到这个广播时，检查 RestrictionsManager 看看当前限制是什么，并对你的 app 的行为做出任何必要的改变。

定义 app 的限制

你的 app 支持任何你想要定义的限制。你在限制文件中声明 app 的限制，在 manifest 中声明限制文件。创建一个限制文件允许其他 app 检查你的 app 提供的限制。企业移动管理（EMM）合作者可以通过 Google Play 接口来读取你的 app 的限制。

为了定义你的 app 的远程配置选项，把以下元素放在你的 manifest 中的 \ 元素里。

```
<meta-data android:name="android.content.APP_RESTRICTIONS"
    android:resource="@xml/app_restrictions" />
```

在 `res/xml` 文件夹中创建一个名为 `app_restrictions.xml` 的文件。该文件的结构在 RestrictionsManager的参考文献中有所 描述。该文件有一个单独的顶级的 `<restrictions>` 元素，这个元素包括一个 `<restriction>` 子元素对应 app 的每一个配置选项。

> 注意：不要创建限制文件的地区化版本。你的 app 只允许有一个限制文件，这样你的 app 在所有地区的限制才会保持一致。

在一个企业环境中，EMM 一般会使用该限制的框架为 IT 管理员生成远程控制台，所以管理员可以远程配置你的 app 。

例如，假设你的 app 可以被远程配置允许或禁止它在蜂窝连接下下载数据。你的 app 就会有一个像这样的 `<restriction>` 元素：

```
<?xml version="1.0" encoding="utf-8"?>
<restrictions xmlns:android="http://schemas.android.com/apk/res/android" >

  <restriction
    android:key="download_on_cell"
    android:title="@string/download_on_cell_title"
    android:restrictionType="bool"
    android:description="@string/download_on_cell_description"
    android:defaultValue="true" />

</restrictions>
```

RestrictionsManager 的参考文献中记载了 `android:restrictionType` 元素所支持的类型。

> 注意：Goole Play for Work 不支持 `bundle` 和 `bundle_array` 限制类型。

使用每个限制的 `android:key` 属性从限制 bundle 中读取它的值。为此，每个限制必须有一个独特的 key 字符串，并且不能被地区化。它必须用一个 string 直接量指明。

> 注意：如在资源地区化所说，在一个产品 app 中，`android:title` 和 `android:description` 应该从地区化资源文件中提取出来。

限制提供者可以询问 app 来找到该 app 可用限制的细节，包括它们的描述文本。限制提供者和企业管理员可以在任何时候，甚至 app 没有在运行的时候，改变它的限制。

检查 app 的限制

当其他 app 改变你的 app 的限制设定时，你的 app 不会被自动通知。反而需要你在 app 启动或恢复的时候检查有哪些限制，并且监听系统 intent 来发现当你的 app 运行的时候限制是否发生改变。

为了知道当前限制设定，你的 app 使用一个 RestrictionsManager 对象。你的 app 应该在以下时候检查当前限制：

- 当 app 启动或者恢复的时候，在它的 onResume() 方法里检查
- 如监听 app 配置的改变中所说，当 app 被提示限制改变的时候

为了获得一个 RestrictionsManager 对象，使用 getActivity() 取得当前 activity，然后调用 activity 的 Activity.getSystemService() 方法：

```
RestrictionsManager myRestrictionsMgr =
    (RestrictionsManager) getActivity()
        .getSystemService(Context.RESTRICTIONS_SERVICE);
```

一旦你有了 RestrictionsManager，你可以通过调用它的 getApplicationRestrictions() 方法取得当前的限制设定：

```
Bundle appRestrictions = myRestrictionsMgr.getApplicationRestrictions();
```

> 注意：方便起见，你也可以用 UserManager 取得当前限制，调用 UserManager.getApplicationRestrictions() 即可。这个方法与 RestrictionsManager.getApplicationRestrictions() 起到完全相同的作用。

getApplicationRestrictions() 方法需要从数据存储区获得数据，所以要尽量少用。不要每次你需要知道当前限制的时候就调用这个方法。你应该只在你的 app 启动或恢复的时候调用，并且缓存所取得的限制 bundle。然后，如监听 app 配置的改变中所说，在你的 app 活动的时候，监听 ACTION_APPLICATION_RESTRICTIONS_CHANGED intent 来发现限制是否改变。

当你的 app 使用 RestrictionsManager.getApplicationRestrictions() 检查限制时，我们建议你检查企业管理员是否把键值对 KEY_RESTRICTIONS_PENDING 设置为 true。如果设置了，你应该阻止用户使用这个 app，并提示他们联系他们的企业管理员。然后，这个 app 应该继续正常运行，注册 ACTION_APPLICATION_RESTRICTIONS_CHANGED 广播。

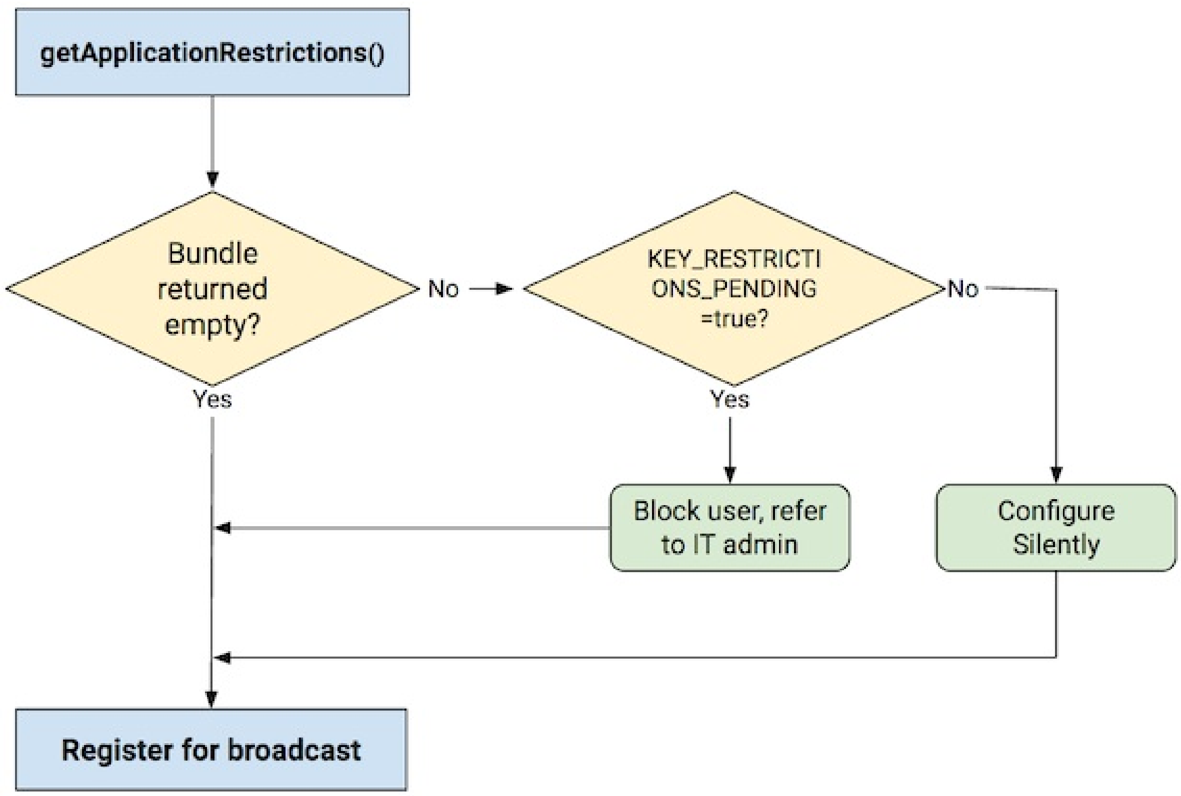


Figure 1. 在注册广播之前检查限制是否暂挂

读取并应用限制

getApplicationRestrictions() 方法返回一个 Bundle，其中包含了被设置的每个限制的键值对。这些值的类型是 `Boolean`，`int`，`String`，`String[]`，`Bundle`，`Bundle[]`。只要你有了限制 Bundle，你就可以用标准的 Bundle 方法针对数据类型来检查当前的限制设置，比如 getBoolean() 或者 getString()。

> 注意：限制 Bundle 为每个被限制提供者显式设置的限制都包括了一个条目。但是，你不能只因为你在限制 XML 文件中定义了一个默认值，就假定这个限制就会在 bundle 里出现。

你基于当前的限制设定，为你的 app 采取合适的行动。比如，如果你的 app 有一个限制架构来指明是否它能在蜂窝连接（就像在定义 app的 限制的例子里一样）中下载，而你发现限制设置为 false，那么你不得不禁止数据下载，除非设备在 Wi-Fi 连接下，正如下面的实例代码所展示的：

```
boolean appCanUseCellular;

if appRestrictions.containsKey("downloadOnCellular") {
    appCanUseCellular = appRestrictions.getBoolean("downloadOnCellular");
} else {
    // here, cellularDefault is a boolean set with the restriction's
    // default value
    appCanUseCellular = cellularDefault;
}

if (!appCanUseCellular) {
    // ...turn off app's cellular-download functionality
    // ...show appropriate notices to user
}
```

> 注意：该限制架构必须向前向后兼容，因为 Google Play for Work 对于每个 app 只给予 EMM 一个版本的限制架构。

监听 app 限制的改变

每当 app 的限制被改变，系统就创建 ACTION_APPLICATION_RESTRICTIONS_CHANGED intent。你的 app 必须监听这个 intent，这样你就能在限制设定改变的时候改变 app 的行为。

> 注意：ACTION_APPLICATION_RESTRICTIONS_CHANGED intent 只发送给动态注册的监听者，而不发送给在 app manifest 里声明的监听者。

以下代码展示了如何为这个 intent 动态注册一个广播接收者：

```
IntentFilter restrictionsFilter =
    new IntentFilter(Intent.ACTION_APPLICATION_RESTRICTIONS_CHANGED);

BroadcastReceiver restrictionsReceiver = new BroadcastReceiver() {
  @Override public void onReceive(Context context, Intent intent) {

    // Get the current restrictions bundle
    Bundle appRestrictions =

    myRestrictionsMgr.getApplicationRestrictions();

    // Check current restrictions settings, change your app's UI and
    // functionality as necessary.

  }

};

registerReceiver(restrictionsReceiver, restrictionsFilter);
```

> 注意：一般来说，当你的 app 中止时不需要被通知限制的改变。相反，这个时候你需要注销你的广播接收者。当 app 恢复时，你首先要检查当前的限制（正如在检查 app 的限制中所讨论的），然后注册你的广播接收者，以保证在 app 活动期间如果有限制改变你会被通知。

创建设备策略控制器

> 编写：zenlynn 原文：http://developer.android.com/training/enterprise/work-policy-ctrl.html

在 Android for Work 的部署中，企业需要保持对员工设备的某些方面的控制。企业需要确保工作相关的信息被加密，并与员工的私人数据分离。企业也可能需要限制设备的功能，例如设备是否被允许使用相机。而且企业也可能需要那些被批准的应用提供应用限制，所以企业可以根据需要关闭或打开应用的功能。

为了处理这些任务，企业开发并部署设备策略控制器应用（以前称为工作策略控制器）。该应用被安装在每一个员工的设备中。安装在每一个员工设备中的控制应用创建了一个企业用户 profile，它可以区别用户的私人账户以访问企业应用和数据。该控制应用同时也是企业管理软件和设备之间的桥梁；当企业需要改变配置的时候就告诉控制应用，然后控制应用适当地为设备和其他应用改变设置。

该课程描述了如何在 Android for Work 的部署中为设备开发一个设备策略控制器。该课程描述了如何创建一个企业用户 profile，如何设置设备策略，以及如何在 managed profile 中为其他运行中的应用进行限制。

> 注意：该课程的内容并不包括在企业控制之下，设备中唯一的 profile 就是 managed profile 的情况。

设备管理概述

在 Android for Work 的部署中，企业管理员可以设置策略来控制员工设备和应用的行为。企业管理员用企业移动管理（EMM）供应商提供的软件设置这些策略。EMM 软件与每一个设备上的设备策略控制器进行通讯。设备策略控制器相应地对每一个私人设备上企业用户 profile 的设置和行为进行管理。

> 设备政策管理器内置于设备管理应用现有的模式中，如设备管理中所说。特别是，你的应用需要创建 DeviceAdminReceiver 的子类，如上述文件所说。

Managed profiles

用户经常想在企业环境中使用他们的私人设备。这种情况可能让企业陷入困境。如果用户使用他们的私人设备，企业不得不担心在这个不受控制的设备上的机密信息（例如员工的电子邮件和通讯录）。

为了处理这种情况，Android 5.0（API 21）允许企业使用 managed profile 建立一个特别的企业用户 profile，或是在 Android for Work 计划中建立一个企业 profile。如果设备有企业 managed profile，该 profile 的设置是在企业管理员的控制之下的。管理员可以选择在这个 profile 之下，什么应用程序可以运行，什么设备功能可以允许。

创建 Managed Profile

要在一个已经有了私人 profile 的设备上创建一个 managed profile，首先得看看该设备是否支持 FEATURE_MANAGED_USERS 系统特性，才能确定该设备是否支持 managed profile：

```
PackageManager pm = getPackageManager();
if (!pm.hasSystemFeature(PackageManager.FEATURE_MANAGED_USERS)) {

    // This device does not support native managed profiles!

}
```

如果该设备支持 managed profile，通过发送一个带有 ACTION_PROVISION_MANAGED_PROFILE 行动的 intent 来创建一个 managed profile。另外要包括该设备的管理包名。

```
Activity provisioningActivity = getActivity();

// You'll need the package name for the WPC app.
String myWPCPackageName = "com.example.myWPCApp";

// Set up the provisioning intent
Intent provisioningIntent =
        new Intent("android.app.action.PROVISION_MANAGED_PROFILE");
intent.putExtra(myWPCPackageName,
        provisioningActivity.getApplicationContext().getPackageName());

if (provisioningIntent.resolveActivity(provisioningActivity.getPackageManager())
         == null) {

    // No handler for intent! Can't provision this device.
    // Show an error message and cancel.
} else {

    // REQUEST_PROVISION_MANAGED_PROFILE is defined
    // to be a suitable request code
    startActivityForResult(provisioningIntent,
            REQUEST_PROVISION_MANAGED_PROFILE);
    provisioningActivity.finish();
}
```

系统通过以下行为响应这个 intent：

- 验证设备是被加密的。如果没有加密，在继续操作之前系统会提示用户对设备进行加密。
- 创建一个 managed profile。
- 从 managed profile 中移除非必需的应用。
- 复制设备策略控制器应用到 managed profile 中，并将设备策略控制器设置为该 profile 的所有者。

如以下实例代码所示，重写 onActivityResult() 来查看部署是否完成。

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {

    // Check if this is the result of the provisioning activity
    if (requestCode == REQUEST_PROVISION_MANAGED_PROFILE) {

        // If provisioning was successful, the result code is
        // Activity.RESULT_OK
        if (resultCode == Activity.RESULT_OK) {
            // Hurray! Managed profile created and provisioned!
        } else {
            // Boo! Provisioning failed!
        }
        return;

    } else {
        // This is the result of some other activity, call the superclass
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

创建 Managed Profile 之后

当 profile 部署完成，系统调用设备策略控制器应用的 DeviceAdminReceiver.onProfileProvisioningComplete() 方法。重写该回调方法来完成启用 managed profile。

通常，你的 DeviceAdminReceiver.onProfileProvisioningComplete() 会执行这些任务：

- 如建立设备政策所述，确认设备遵守 EMM 的设备策略
- 使用DevicePolicyManager.enableSystemApp()) 来启动管理员在 managed profile 中允许使用的任何系统应用

- 如果设备使用 Google Play for Work，用 AccountManager.addAccount() 在 managed profile 中添加 Google 账号，管理员就能往设备中安装应用了。

一旦你完成了这些任务，调用设备策略管理器的 setProfileEnabled() 方法来激活 managed profile：

```
// Get the device policy manager
DevicePolicyManager myDevicePolicyMgr =
        (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);

ComponentName componentName = myDeviceAdminReceiver.getComponentName(this);

// Set the name for the newly created managed profile.
myDevicePolicyMgr.setProfileName(componentName, "My New Managed Profile");

// ...and enable the profile
manager.setProfileEnabled(componentName);
```

建立设备策略

设备策略管理器应用负责实行企业的设备策略。例如，某个企业可能需要在输错一定次数的设备密码后锁定所有设备。该控制器应用需要 EMM 查出当前的策略是什么，然后使用设备管理 API 来实行这些策略。

更多关于如何实行设备策略的信息，请查看设备管理指南。

实行应用限制

企业环境可能需要那些批准的应用实现安全性或功能限制。应用开发人员必须实现这些限制，并声明由企业管理员使用，如实现应用的限制所说。设备政策管理器接收来自企业管理员改变的限制，并将这些限制的改变传送给相关应用。

例如，某个新闻应用有一个控制应用是否允许在蜂窝网络下下载视频的限制设定。当 EMM 想要禁用蜂窝下载，它就给控制器应用发送通知。于是控制器应用转而通知新闻应用限制设定被改变了。

> 注意：本文档涵盖了设备策略管理器应用如何改变 managed profile 中其他应用的限制设定。关于设备策略管理器应用如何与 EMM 进行通讯的细节并不在本文档的范围之内。

为了改变一个应用的限制，调用 DevicePolicyManager.setApplicationRestrictions() 方法。该方法需要传入三个参数：该控制器应用的 DeviceAdminReceiver，限制被改变的应用的包名，以及包含了你想要设置的限制的 Bundle。

例如，假设 managed profile 中有一个应用包名是 `"com.example.newsfetcher"` 。该应用有一个布尔型限制可以被配置，key 是 `"downloadByCellular"` 。如果这个限制被设置为 `false` ，该应用在蜂窝网络下就不能下载数据，它必须使用 Wi-Fi 网络代替。

如果你的设备策略管理器应用需要关掉蜂窝下载，它首先要取得设备策略服务对象，如上文所说。然后集合一个限制 bundle 并将该 bundle 传入 setApplicationRestrictions()：

```
// Fetch the DevicePolicyManager
DevicePolicyManager myDevicePolicyMgr =
        (DevicePolicyManager) thisActivity
                .getSystemService(Context.DEVICE_POLICY_SERVICE);

// Set up the restrictions bundle
bundle restrictionsBundle = new Bundle();
restrictionsBundle.putBoolean("downloadByCellular", false);

// Pass the restrictions to the policy manager. Assume the WPC app
// already has a DeviceAdminReceiver defined (myDeviceAdminReceiver).
myDevicePolicyMgr.setApplicationRestrictions(
        myDeviceAdminReceiver, "com.example.newsfetcher", restrictionsBundle);
```

> 注意：该设备策略服务将限制改变传递给了你所指定的应用。然而，实际是由应用来执行该限制。例如，在这个情况中，该应用要负责禁用它本身的使用蜂窝网络下载视频的功能。设置限制并不能让系统强制在应用上实现限制。更多信息，请查看实现应用的限制。

Android交互设计

> 编写:kesenhoo - 原文:http://developer.android.com/training/best-ux.html

These classes focus on the best Android user experience for your app. In some cases, the success of your app on Android is heavily affected by whether your app conforms to the user's expectations for UI and navigation on an Android device. Follow these recommendations to ensure that your app looks and behaves in a way that satisfies Android users.

Designing Effective Navigation

How to plan your app's screen hierarchy and forms of navigation so users can effectively and intuitively traverse your app content using various navigation patterns.

Implementing Effective Navigation

How to implement various navigation patterns such as swipe views, a navigation drawer, and up navigation.

Notifying the User

How to display messages called notifications outside of your application's UI.

Adding Search Functionality

How to properly add a search interface to your app and create a searchable database.

Making Your App Content Searchable by Google

How to enable deep linking and indexing of your application content so that users can open this content directly from their mobile search results.

设计高效的导航

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/index.html

设计开发 App 的起初步骤之一就是决定用户能够在App上看到什么和做什么。一旦你知道用户在App上和哪种内容互动，下一步就是去设计容许用户在 App 的不同内容块间切换，进入，回退的交互。

本课程演示如何为你的应用规划出高标准的界面层次，然后为它选择适宜的导航形式来允许用户高效而直观的浏览内容。按粗略的先后顺序,每堂课涵盖Android应用导航交互设计过程中的不同阶段。学过这些课之后，你应该可以应用这些列出的方法和设计范例到你自己的应用中，为你的用户提供一致的导航体验了。

Lessons

- 规划界面和他们之间的关系

  学习如何选择你应用应该包含的界面。并且学习如何选择其他界面可直达的界面。这节课介绍了一个假想的新闻应用为以后课程作例子。

- 为多种大小的屏幕进行规划

  学习如何在大屏设备上组合相关界面来优化用户可视界面空间。

- 提供向下和横向导航

  学习容许用户深入某一层或者在内容层次间横跨的技巧。而且学习一些特定导航 UI 元素在不同情景下的优缺点和最佳用法。

- 提供向上和历史导航

  学习如何容许用户在内容层级向上导航。并且学习 Back 键和历史导航的最佳做法，也即导航到和层次无关的之前的画面。

- 综合：设计样例 **App**

  学习如何创建界面的 Wireframe（线框图，模糊的图形模型）来代表新闻应用基于设想信息模型的界面。这些 Wireframe 利用上述课程讨论的导航元件来展示直观高效导航。

规划界面和他们之间的关系

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/screen-planning.html

多数 App 都有一种内在的信息模型，它能被表示成一个用对象类型构成的树或图。更浅显的说，你可以画一个有不同类型信息的图，这些信息代表用户在你 App 里用户与之互动的各种东西。软件工程师和数据架构师经常使用实例-关系图（Entity-Relationship Diagram，ERD）描述一个应用的信息模型。

让我们考虑一个让用户浏览一群已分类好的新闻事件和图片的应用例子。这种 App 一个可能的模型如下 ERD 图。

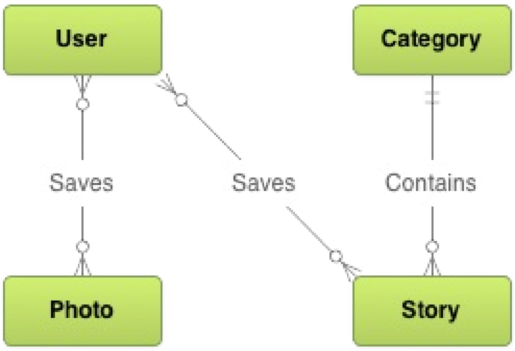


Figure 1. 新闻应用例子的实例关系图

创建一个界面列表

一旦你定义了信息模型，你就可以开始定义那些能使用户在你的 App 中有效地发掘，查看和操作数据的上下文环境了。实际上，其中一种方法就是确定供用户导航和交互数据所需的 界面完备集（归纳了所有界面的集合）。但我们实际发现的界面集合应该根据目标设备变化。在设计过程中早点考虑到这点很重要，这样可以保证程序可以适应运行环境。

在我们的例子中，我们想让用户查看，保存和分享分类好了的新闻和图片。下面是涵盖了这些用例的界面完备列表。

- 用来访问新闻和图片的 Home 或者 "Launchpad" 画面
- 类别列表
- 某个分类下的新闻列表
- 新闻详情 View （在这里我们可以保存和分享）
- 图片列表，不分类
- 图片详情 View （在这里我们可以保存和分享）

- 所有保存项列表
- 图片保存列表
- 新闻保存列表

图示界面关系

现在我们可以定义界面间的有向关系了。一个从界面 *A* 指向另一个界面 *B* 的箭头表示通过用户在画面 *A* 的某个交互动作可直达画面 *B* 。一旦我们定义了界面集和他们之间的关系，我们可以将他们一起全部表示在一张界面图中了：

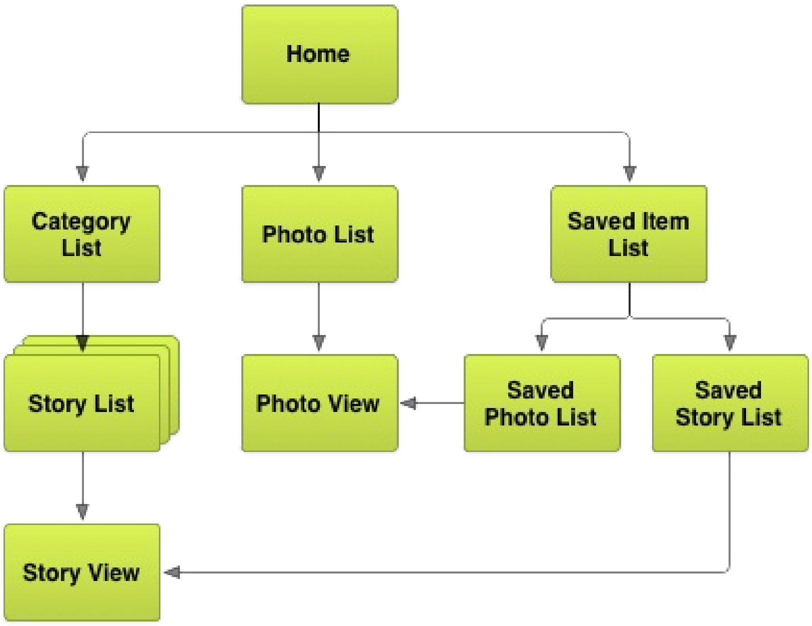


Figure 2. 新闻应用例子的界面完备Map

如果之后我们想允许用户提交新闻事件或者上传图片，我们可以在图中加额外的界面。

脱离简陋设计

这时，我们可以据这张完备的界面图设计一个功能完备应用了。可以由列表和导向子界面的按钮构成一个简单的UI：

- 导向不同页面的按钮（例如，新闻，图片，保存的项目）
- 纵向列表表示集合（例如，新闻列表，图片列表，等等）
- 详细信息（例如，新闻 View ，图片 View ，等等）

但是，你可以利用屏幕组合技术和更高深导航元素以一种更直观，设备更理解的方式呈现内容。下节课，我们探索屏幕组合技术，比如为平板而生的多视窗（Multi-pane）布局。之后，我将深入讲解更多不同的 Android 常见导航模式。

下节课：规划多种触屏大小

为多种大小的屏幕进行规划

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/multiple-sizes.html

虽然上节中的界面完备图在手持设备和相似大小设备上可行，但并不是和某个设备因素绑死的。Android应用需要适配一大把不同类型的设备，从3"的手机到10"的平板到42"的电视。这节课中我们探讨把完备图中不同界面组合起来的策略和原因。

> **Note:** 为电视设计应用程序还需要注意其他的因素，包括互动方式（就是说，它没触屏），长距离情况下文本的可读性，还有其他的。虽然这个讨论在本课范畴之外，你仍然可以在 Google TV 文档的设计模式中找到有关为电视设计的信息。

用多视窗布局（Multi-pane Layout）组合界面

> 多视窗布局（**Multi-pane Layout**）设计
>
> 设计指南请阅读 Android 设计部分的多视窗布局。

3 到 4英寸的屏幕通常只适合每次展示单个纵向内容视窗，一个列表，或某列表项的具体信息，等等。所以在这些设备上，界面通常对映于信息层次上的某一级（类别 → 列表 → 详情）。

更大的诸如平板和电视上的屏幕通常会有更多的可用界面空间，并且他们能够展示多个内容视窗。横屏中，视窗从左到右以细节程度递增的顺序排列。因常年使用桌面应用和网站，用户变得特别适应大屏上的多视窗。很多桌面应用和网站提供左侧导航视窗，或者使用总/分（master/detail）两个视窗布局。

为了符合这些用户期望，通常很有必要为平板提供多个信息视窗来避免留下过多空白或无意间引入尴尬的交互，比如 10 x 0.5" 按钮。

下面图例示范了当把 UI 设计迁移到更大的布局时出现的一些问题，并且展示了如何用多视窗布局来处理这些问题：



图 **1.** 大横屏使用单视窗导致尴尬的空白和过长行。



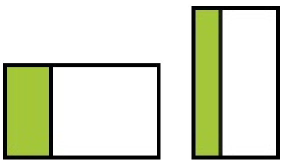
图 **2.** 横屏多视窗布局产生更好的视觉平衡，更大的效用和可读性。

> 实现提醒： 当决定好了区分使用单视窗布局和多视窗布局的屏幕大小基准线后，你就可以为不同屏幕大小区间（例如 `large/xlarge` ）或最低屏幕宽度（例如 `sw600dp` ）提供不同的布局了。
>
> 实现提醒： 单一界面被实现为 Activity 的子类, 单独的内容视窗则可实现为 Fragment 的子类。这样最大化了跨越不同结构因素和不同屏幕内容的代码复用。

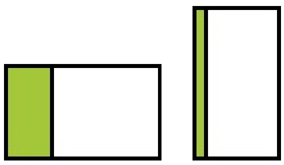
为不同平板方向设计

虽然现在我们还没有开始在我们的屏幕上排布 UI 元素，但现在很是时候来考虑下我们的多视窗界面如何适配不同的设备方向了。多视窗布局在横屏时表现的非常棒，因为有大量可用的横向空间。然而，在竖屏时，你的横向空间被限制了，所以你需要为这个方向设计一个单独的布局。

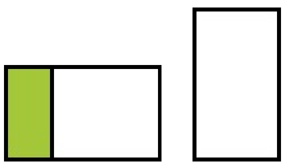
下面是一些创建竖屏布局的常见策略：

- 伸缩 

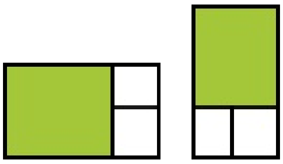
  最直接的策略就是简单地伸缩每个视窗的宽度来最好地在竖屏下的呈现内容。视窗可设置固定宽度或占可用界面宽度的一定比例。

- 展开/折叠 

  伸缩策略的一个变种就是在竖屏中折叠左侧视窗的内容。当遇到总/分（master/detail）视窗中左侧（master）视窗包含易折叠列表项时，这个策略很有效。以一个实时聊天应用为例。横屏中，左侧列表可能包含聊天联系人的照片，姓名和在线状态。在竖屏中，横向空间可以将通过隐藏联系人姓名而且只显示照片和在线状态的提示图标的方式来折叠。也可以选择性的提供展开控制，这种控制允许用户展开左侧视窗或反向操作。

- 显示/隐藏 

  这个方案中，左侧视窗在竖屏模式下完全隐藏。然而，为了保证你界面的功能等价性，左侧视窗必须功能可见（比如，添一个按钮）。通常适合在 Action Bar 使用 *Up* 按钮（详见Android设计的模式文档）来展示左侧视窗，这将在之后讨论。

- 堆叠 

  最后的策略就是在竖屏时垂直地堆放你一般横向排布的视窗。当你的视窗不是简单的文本列表，或者当有多个内容模块与基本内容视窗同时运行时，这个策略很奏效。但是当心使用这个策略时出现上面提到的尴尬的空白问题。

组合界面图中的界面

既然现在我们能够通过提供大屏设备上的多视窗布局来组合单独的界面，那么就让我们把这个技术应用到我们上节课界面完备图上吧，这样我们应用的界面层次在这类设备上变得更具体了：

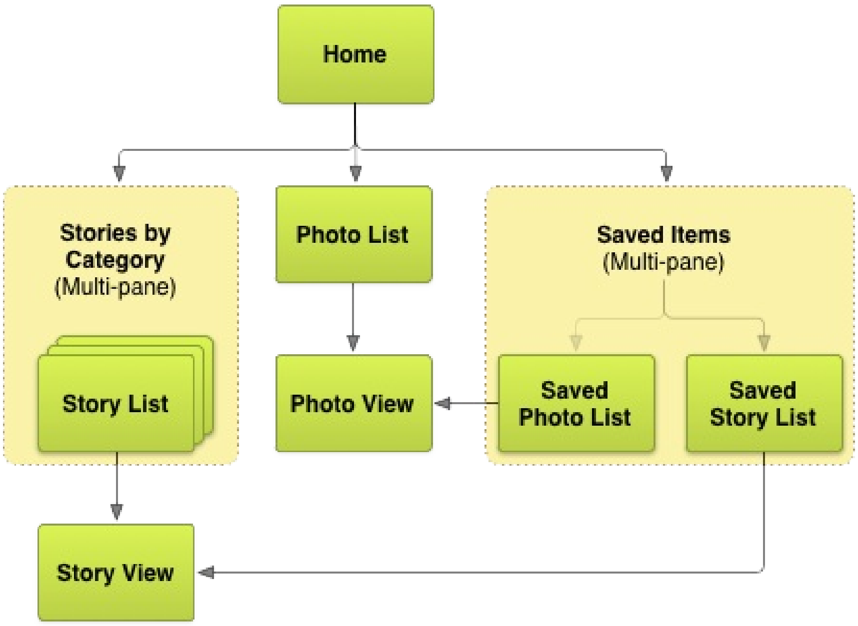


Figure 3. 更新后新闻应用例子的界面完备Map

下节课我们将讨论 向下 和 横向 导航，并且探讨更多方法来组合界面使能最大化应用 UI 的直观性和内容获取速度。

下一节：提供向下和横向导航

提供向下与横向导航

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/descendant-lateral.html

一种提供查看应用整体界面结构的方式就是显示层级导航。这节课我们讨论 向下导航，它允许用户进入子界面。我们还讨论 横向 导航，它允许用户访问同级界面。

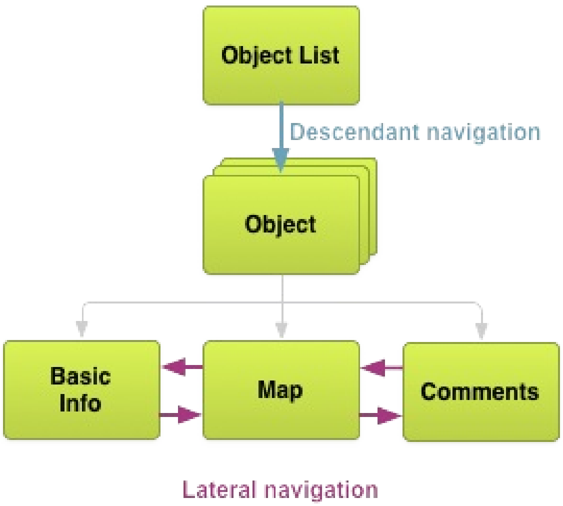


Figure 1. 向下和横向导航

有两种同级界面：容器关联和区块关联界面。容器关联（*Collection-related)* 界面展示由父界面放入同个容器里地那些条目 。区块关联*(Section-related)* 界面展示父界面不同部分的信息，例如：一个部分可能展示某对象的文字信息，可是另一个部分则提供对象地理位置的地图。一个父界面的区块关联界面数量通常较少。

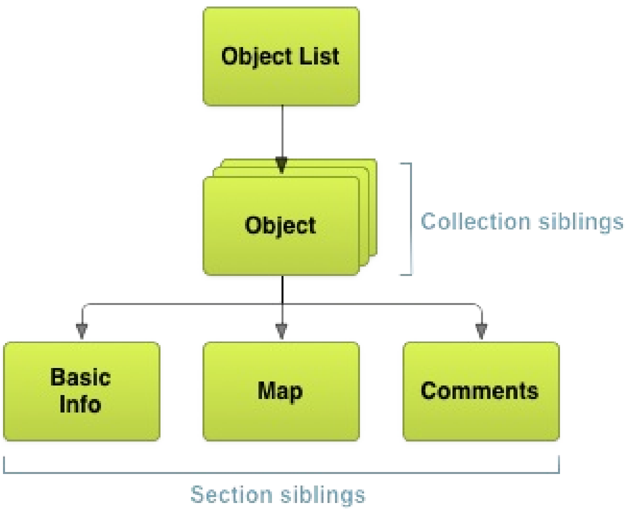


Figure 2. 容器关联子界面和区块关联子界面。

向下和横向导航可用List（列表），Tab（标签）或者其他 UI 模式来实现。 *UI* 模式, 与软件设计模式很类似，是重复交互设计问题的一般化解决方案。下几章，我们将探究一些常用的横向导航模式。

Button和简单的控件

> **Button设计**
>
> 设计指南请阅读 Android 设计文档的Button指导

对于区块关联的界面，最直接和熟悉的导航界面就是提供可触或键盘可得焦点的控件。例如，Button，固定大小的 List View 或 文本链接，虽然后者不是一个触屏导航的理想 UI 元素。一旦点选了这些控件，子界面被打开，完全替代当前上下文环境（屏幕）。Button或其他简单地控件很少被用来呈现容器中的项目。

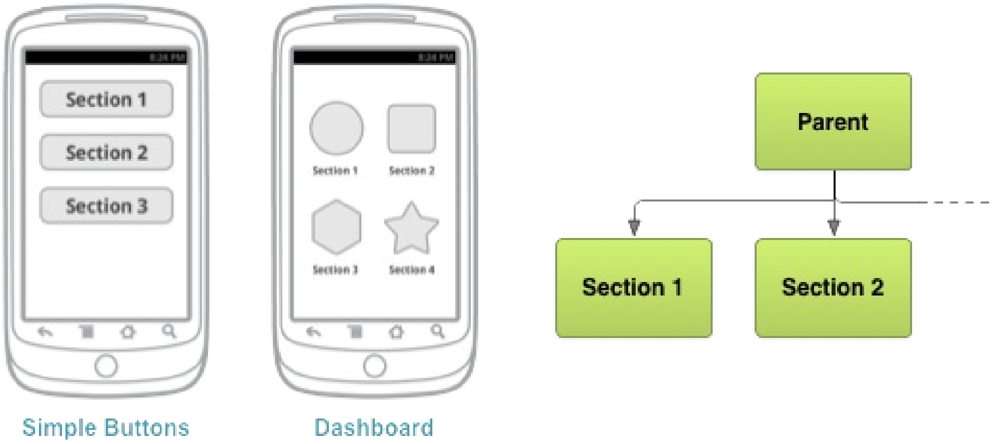


Figure 3. Button导航模式例子和对应界面图。Dashboard 模式见下文。

Dashboard（操作面板）模式是一种一般以Button为主来获取不同应用划分模块的模式。一个dashboard就是个大图标Button表格，它表示了父界面绝大部分内容。这个表格通常是2、3行或列，取决于 App 的顶层划分。此模式展示全部区块的视觉效果非常丰富。巨大的触摸控件也让 UI 特别好使。当每个区块都同等重要时，Dashboard模式最好用。然而，这个模式在大屏上效果不佳，他让用户直接获取 App 内容时多走了一步弯路。

还有更多套用了各种其他 UI 模式来提升内容即得性和独特的展示效果，但仍保持着直观特点的高级 UI 模式。

Lists, Grids, Carousels, and Stacks

> **List 和 Grid List 设计**
>
> 设计指南请阅读 Android 设计文档的Lists和Grid Lists指导。

对于容器关联的界面，特别是文字信息，垂直滑动列表通常是最直接最熟悉的做法。对于视觉更丰富的内容（例如，图片，视频），可用垂直滑动的 Grid，水平滚动的 List（有时被叫做 Carousel），或 Stack（有时叫做卡片，Card）来代替。这些 UI 元素通常用在呈现容器内的条目，或大量子界面最好，而不是零星的毫无关联的同级子界面。

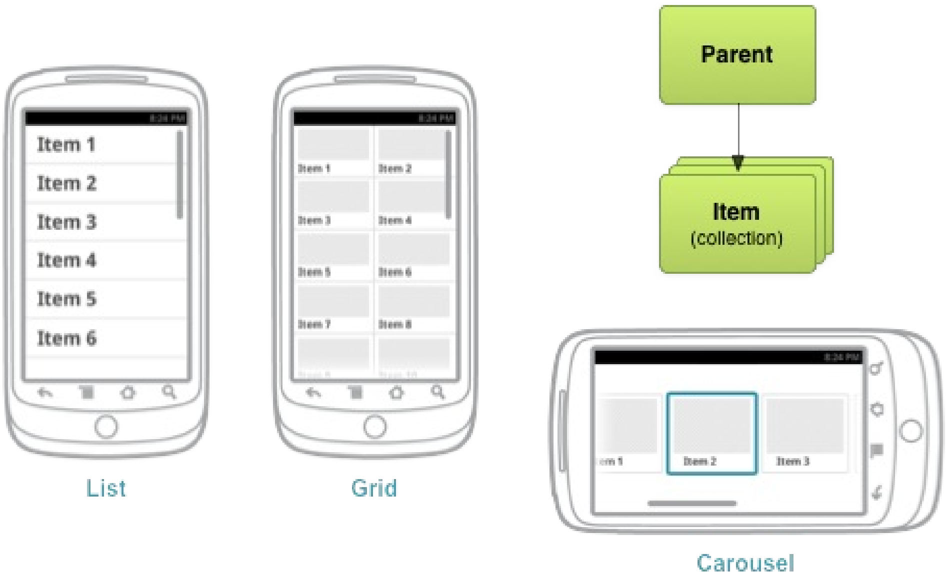


Figure 4. 控件例子和对应界面图

这个模式还有些问题。深层列表导航常常叫 drill-down（钻井）列表导航，它的list层层嵌套。这种导航笨拙低效。获得某块内容需要点击多次，带给用户很差的体验，特别是活跃用户。

使用纵向list也可能带来尴尬的用户交互，并且如果list条目简单地的拉伸话也可能用不好大屏空白。解决方法就是提供额外的信息，例如用文字汇总填充那些可用的水平空间。或者在左右添加个视窗。

Tabs（标签）

> **Tab 设计**
>
> 设计指南请阅读 Android 设计文档的Tab指导

Tab是非常流行的横向导航。这个模式允许组合同级界面，就是说tab可嵌入原本可能成为另一个界面的子界面内容。Tab适合用在小量的区块关联界面。

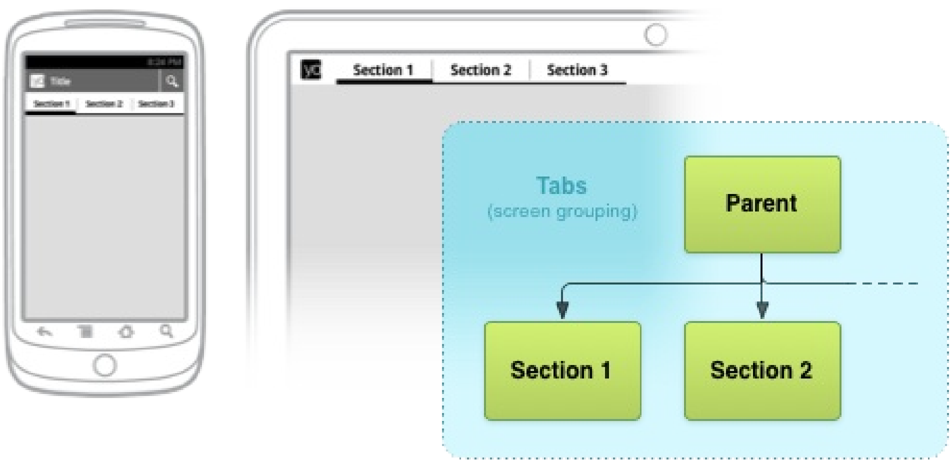


Figure 5. 手机和平板导航例子和对应界面图

几个使用Tab时的最佳做法。Tab在关联界面种应该一直存在，只有指定内容区域发生改变，并且tab提示在任何时候都可用。此外，tab切换不能算作历史。例如，如果用户从 Tab *A* 切换到 Tab *B*，按 *Back* 按钮（详情看下节）不该重选 Tab *A*。Tab通常水平排布，可是有时其他tab展现形式，例如Action bar（详见Android 设计的模式章节）的下拉菜单，也是可以的。最后，最重要的是，tab应该在界面顶端和内容对应。

tab导航相对于list和button导航，有很多即得的优点：

- 既然只有一个初始时既选的活动tab，用户能立即从界面获取tab的内容。
- 用户可在相关界面内快速导航，不用重新访问父界面。

> 注意：当切换Tab时，保证立即切换很重要。不要加载时弹个确认对话框来阻塞tab的访问。

导致这个模式被批评常见的原因就是必须从展示内容的屏幕空间分一些给tab提示栏。但是结果还能接受，权衡一般都向使用此模式的方向倾斜。你可以随意个性化你的tab提示栏，加点文字或图标什么的让纵向空间合理利用。但是调整tab宽度时，请确保tab够大到能让人无误点击。

水平分页（**Swipe View**）

> **Swipe View** 设计
>
> 设计指南请阅读 Android 设计文档的Swipe View指导

另一种横向导航的模式就是水平分页，也叫做 Swipe View。这个模式在容器关联的同级界面上最好用，例如类别列表（世界，金融，技术和健康新闻）。就像Tab，这个模式也允许组合界面，这样父界面就能在布局内嵌入子界面的内容。

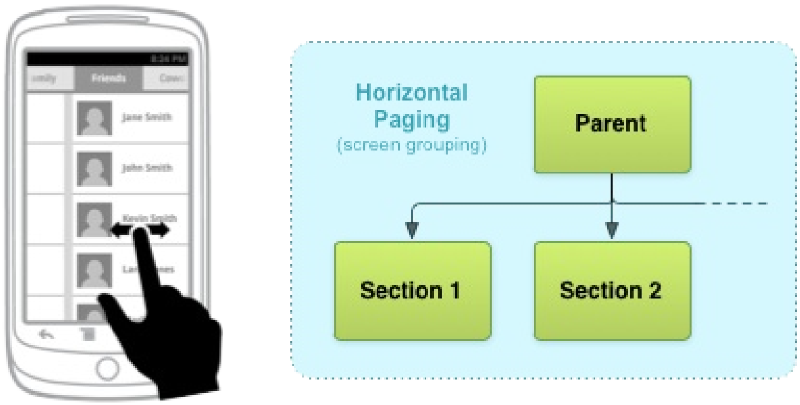


Figure 6. 水平分页导航例子和对应界面图

在水平分页 UI 中，一次只展示一个子界面（这儿叫页，*page*）。用户能通过触摸屏幕然后按想要访问相邻页面的方向拖拽导航到同级界面。为补充这种手势交互通常由另一种 UI 元素提示当前页和可访问页。这样能帮助用户发觉内容并且也提供了更多的上下文环境信息给用户。当为区块关联的同级模块使用这种模式的水平导航时，这个做法很有必要。这些提示界面元素的例子包括点标（tick mark），滑动标注（scrolling label）和标签（tab）：

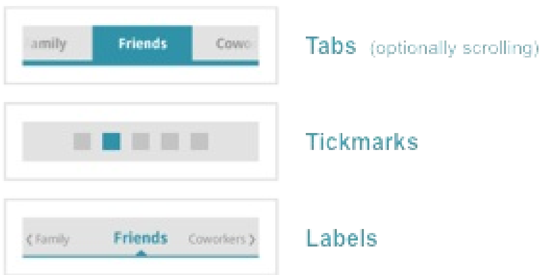


Figure 7. 搭配分页的 UI 元件。

当子界面包含水平平移视图时（例如地图）也最好避免使用这种模式，因为这些冲突的交互会威胁你界面的易用性。

此外，对于同级关联界面，如果内容类型具有一定相似性而且同级界面数量较少时，水平分页再适合不过了。就这一点，这个模式可以和tab一起用。tab放在内容上方来最大化界面直观性。对于容器关联界面，当界面间有天然的顺序时，水平分页是最符合直觉的，例如页面代表连续的日历日。对于无穷无尽的数据，特别是双向都有内容数据，分页机制效果非常棒。

下节课，我们讨论在内容层级中允许用户往上和回退到之前访问界面的导航的机制。

下节课：提供向上和时间导航

提供向上导航与历史导航

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/ancestral-temporal.html

既然现在我们能进入应用界面某个层级，我需要提供一个方法来在层级里向上导航到父亲或祖先界面中。此外，我们应该保证通过 *Back* 按钮来回退历史导航记录。

> 回退/向上导航设计
>
> 设计指南请阅读 Android 设计文档的Navigation模式指导

支持历史导航：*Back*

历史导航，或者说在历史的界面间导航，在 Android 系统中由来已久。不论其他状态如何，所有 Android 用户都期望 *Back* 按钮能带他们回到之前的界面。历史界面集全都以用户的 Launcher 应用为基础（电话的 “Home” 键）。也就是说，按下 *Back* 键足够多次数后你应该回到 Launcher，之后 *Back* 键不做任何事情。

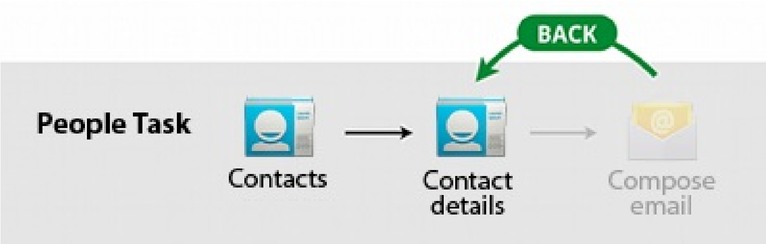


Figure 1. 从 Contacts（联系人）app中进入电子邮件 app 然后按 *Back* 键的行为

应用自身通常不必考虑去管理 *Back* 按钮。系统自己自动处理 *task 和 back H1H2H3H4 stack*（回退栈），或者叫历史界面列表。 *Back* 按钮默认反向访问界面列表，然后当按钮被按下时从列表中移除当前界面。

但是总是有一些你可能需要重写 *Back* 行为的例子。比如，你屏幕包含一个嵌入的网页浏览器，在这个浏览器中你的用户可和页面元件进行交互来在网页间导航。你可能希望当用户按下设备的 *Back* 键时触发嵌入浏览器的默认 *back* 操作。当到达了浏览器内部历史的起始点，你就应该遵从系统 *Back* 按钮的默认行为了。

提供向上导航：*Up* 和 *Home*

Android 3.0 之前，最常见的向上导航的形式以 *Home* 表示。大体上是以在设备 *Menu* 按钮里提供一个 *Home* 的可选项这样的方法来实现，或者 *Home* 按钮出现在屏幕的左上角作为 Action Barbar（详见Android 设计的模式章节）的一个组件。当选中 *Home* 后，用户被带到界面层级的顶层，通常被叫做应用的主界面。

提供对程序主界面的直接访问能带给用户一种舒适感和安全感。无论位于应用程序何处，如果你在 App 中迷路了，你可以点选 *Home* 然后回到那熟悉的主界面。

Android 3.0 引入了 *Up* 记号，它被展示在了 Action Bar 上代替了上述的 *Home* 按钮。点击 *Up*，用户将被带入到结构中的父界面。这个导航操作通常就是进入前一个界面（就像之前 *Back* 按钮讨论中描述的一样），但是并不是永远都这样。因此，开发者必须保证 *Up* 对于每个界面都会导航到某个既定的父亲界面。

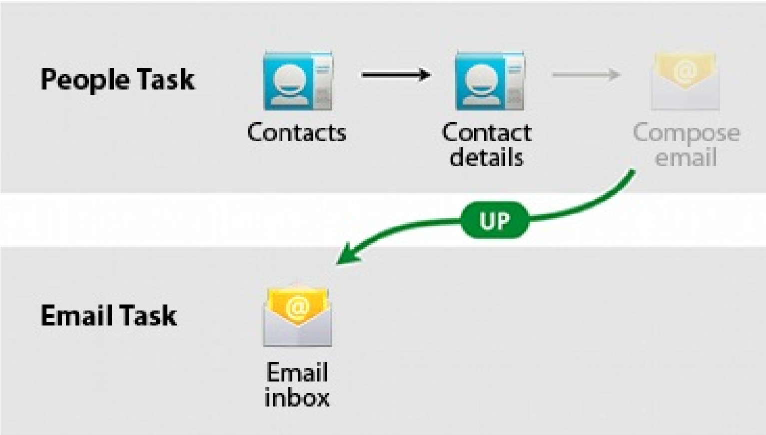


Figure 2. 从联系人 App 中进入电子邮件 App 然后按 *Up* 导航的行为

某些情况下，*Up* 适合执行某个行为而非导航到一个父亲节点。以 Android 3.0 平板上的 Gmail 应用为例。当查看一封邮件的对话时把设备平放，对话列表和对话详情将并排显示。这是一种之前课程中的父、子界面组合。然而，当竖屏查看邮件对话时，只有对话详情被显示。*Up* 按钮被用来使父视窗滑入屏幕显示。当左侧视窗可见时再按一次 *Up* 按钮，单个对话便回到全屏的对话列表中。

> 实现提醒： 实现 *Home* 或 *Up* 导航的最佳做法就是保证清除back stack中的子界面。对于 *Home*，Home 界面是唯一留在back stack中的界面。对于 *Up* 导航，当前界面也应该从back stack中移除，除非 *Back* 在不同界面层级间导航。你可以将 FLAG_ACTIVITY_CLEAR_TOP和FLAG_ACTIVITY_NEW_TASK这两个 Intent 标记一起使用来实现它。

最后一节课中，我们应用现在为止所有课程中讨论的概念来为我们新闻应用例子创建交互设计 Wireframe（线框图）。

下节课：综合：设计我们的样例 App

综合：设计我们的样例 App

> 编写:XizhiXu - 原文:http://developer.android.com/training/design-navigation/wireframing.html

现在我们对导航模式和界面组合技术有了深入的理解，是时候应用到我们的界面上了。让我再看看我们第一节课上提到的新闻应用的界面完备图：

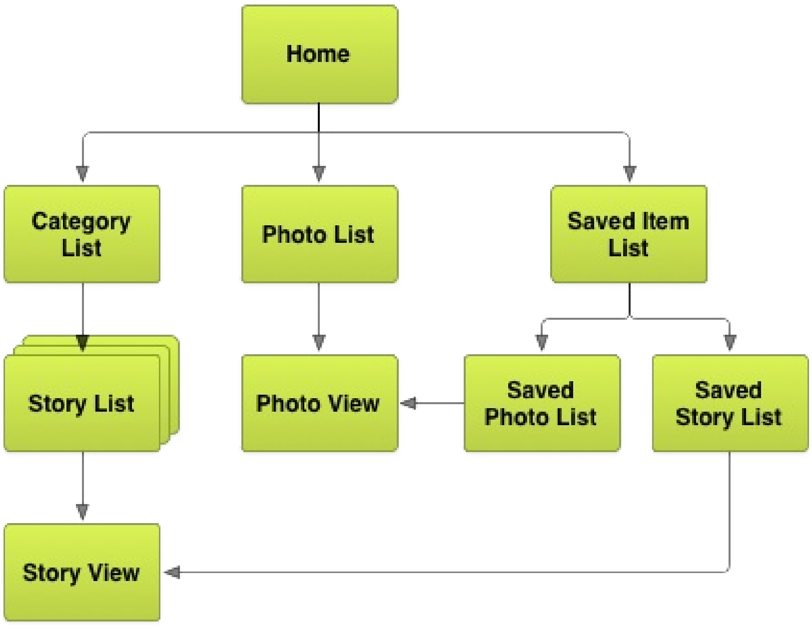


Figure 1. 新闻应用例子的界面完备集

我们下一步得去我们前几节讨论的导航模式选择，然后应用到这个界面图中。这样就能最大化导航速度并且最少化获取内容的点击次数，但又能参考 Android 做法来保证界面的直观性和一致性。此外，我们也需要根据我们不同目标设备的参数做出不同的决定。为方便，我们集中讨论平板和手持设备。

选择模式

首先，我们二级界面（新闻类别列表 ，图片列表 和 保存列表）可用 Tab 组合在一起。注意到我们不必使用水平排列的 Tab；某些情况下下拉菜单可作为合适的替代品，特别在手机这种窄屏设备上。在手机上，我们能用 Tab 把 图片保存列表 和 新闻保存列表 组合到一起，或在平板上用多个纵向排列的内容视窗。

最后，让我们看看如何展示新闻。第一个简化不同新闻类别间导航的选项：使用水平分页，然后再在滑动区域上添加一组标签来提示当前可见和临近的新闻类别。对于平板横屏，我们可以进一步地展示能水平分页的 新闻列表 界面作为左边的视窗，并且把 新闻详情 *View* 界面作为基础内容视窗放在右边。

下图分别表示在手持设备和平板上应用了这些导航模式后的新界面图。

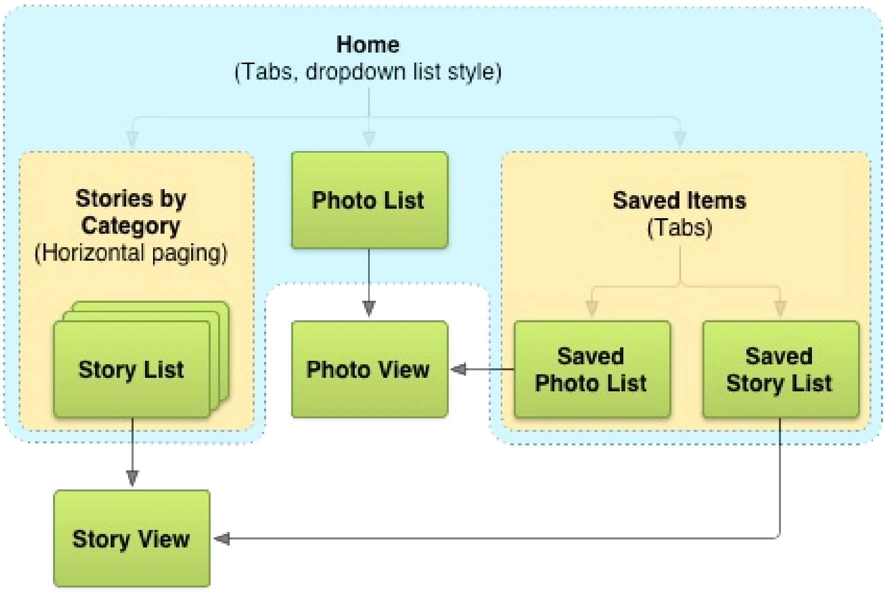


Figure 2. 手持设备上新闻应用例子的最终界面集

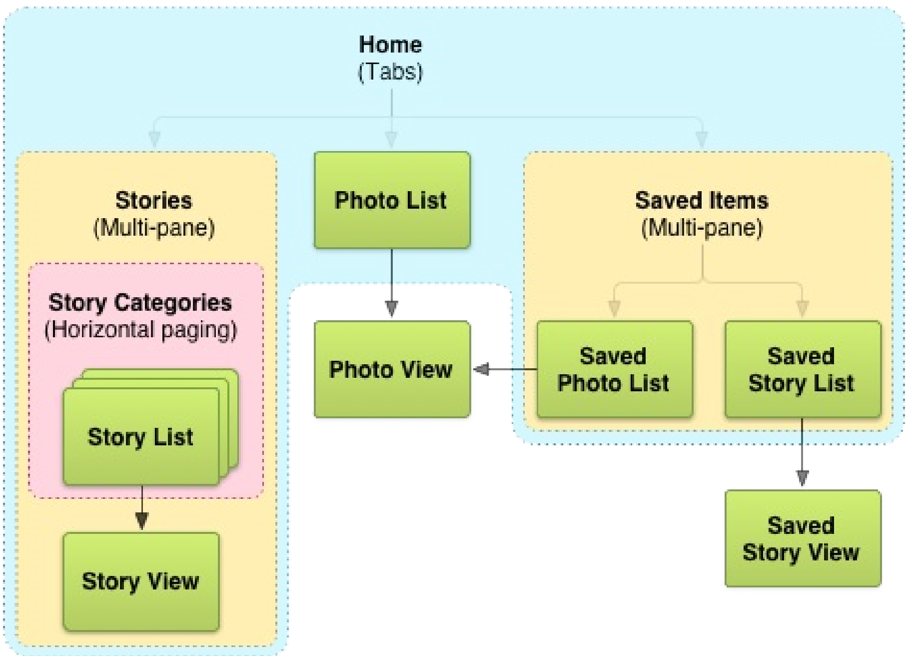


Figure 3. 平板上新闻应用例子的最终界面集，横屏

至此，得好好考虑下界面图的衍化了，以免我们选择的模式实际上用不了（比如当你画应用界面布局的草图时）。下面有个为平板衍化的界面图样例，它并排展示不同类别的 新闻列表，但是 新闻详情*View* 保持独立。

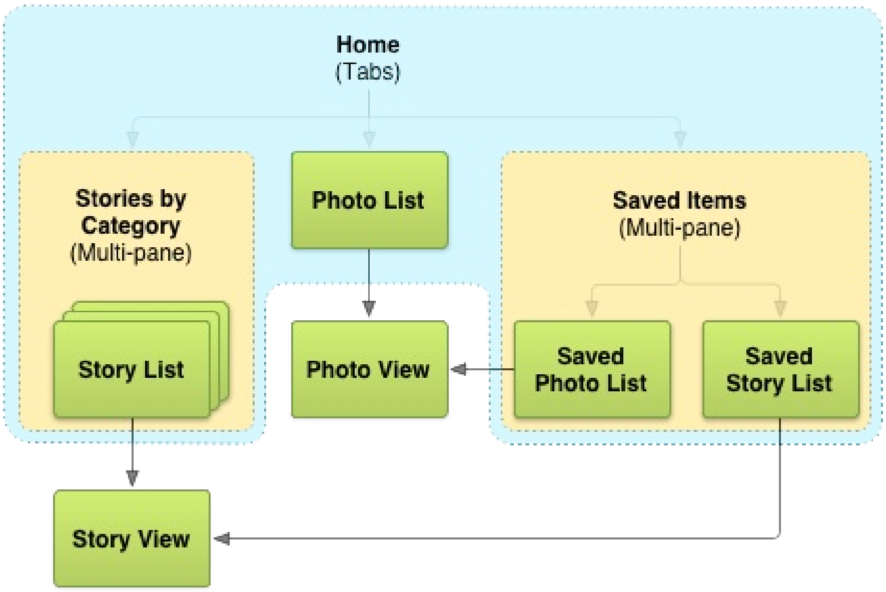


Figure 4. 平板上新闻应用例子的最终界面集，竖屏

画草稿

Wireframing就是设计过程中你开始排布界面的那步。发挥你的创造性，想想怎么排列这些 UI 元件来帮助你的用户在你的 App 中导航。这时你要记住细枝末节是不重要的（别去想着做个实物）。

最简单快速的起步方法就是用纸笔手画你界面。一旦你开始画，你会发现在你原本的界面图或在你决定使用的模式中有很多实际的问题。某些情况下，模式理论上能很好的解决特定设计问题，但实际上他们可能失效并且给视觉交互添乱（例如，界面上出现了两行 Tab）。如果那样，探索下其他的导航模式，或在选择的模式上做点变化，来让你的草稿更优。

当你对初稿满意后，继续用一些软件画你的数字wireframe吧，例如：Adobe® Illustrator，Adobe® Fireworks，OmniGraffle 或者 向量图工具。选择画图工具时，考虑以下特性：

- 能画体现交互的 wireframe 么？像Adobe® Fireworks就能提供这个功能。
- 有界面“大师”功能（允许不同界面的视觉元素重用）？例如，Action Bar必须在你应用的每个界面都出现。
- 学习曲线怎样？专业向量图工具可能有个陡峭的学习曲线（越学越难），但有些功能小巧的 wireframing 设计工具可能更适合这个任务。

最后，XML 布局编辑器，Android 开发工具包（ADT）里面的一个 Eclipse 插件，经常被用来画草图原型。但是，你应当贯注于高质量的布局而非细节视觉设计。

创建数字草图

在纸上画完草图并且选择好一款心仪的数字wireframing工具后，你可以创建一个数字wireframe作为你应用视觉设计的起点。下面就是一些我们新闻客户端wireframe例子，他们和我们之前的界面图一一对应。



Figure 5. 新闻客户端手机竖屏Wireframe样例（下载 SVG 图）

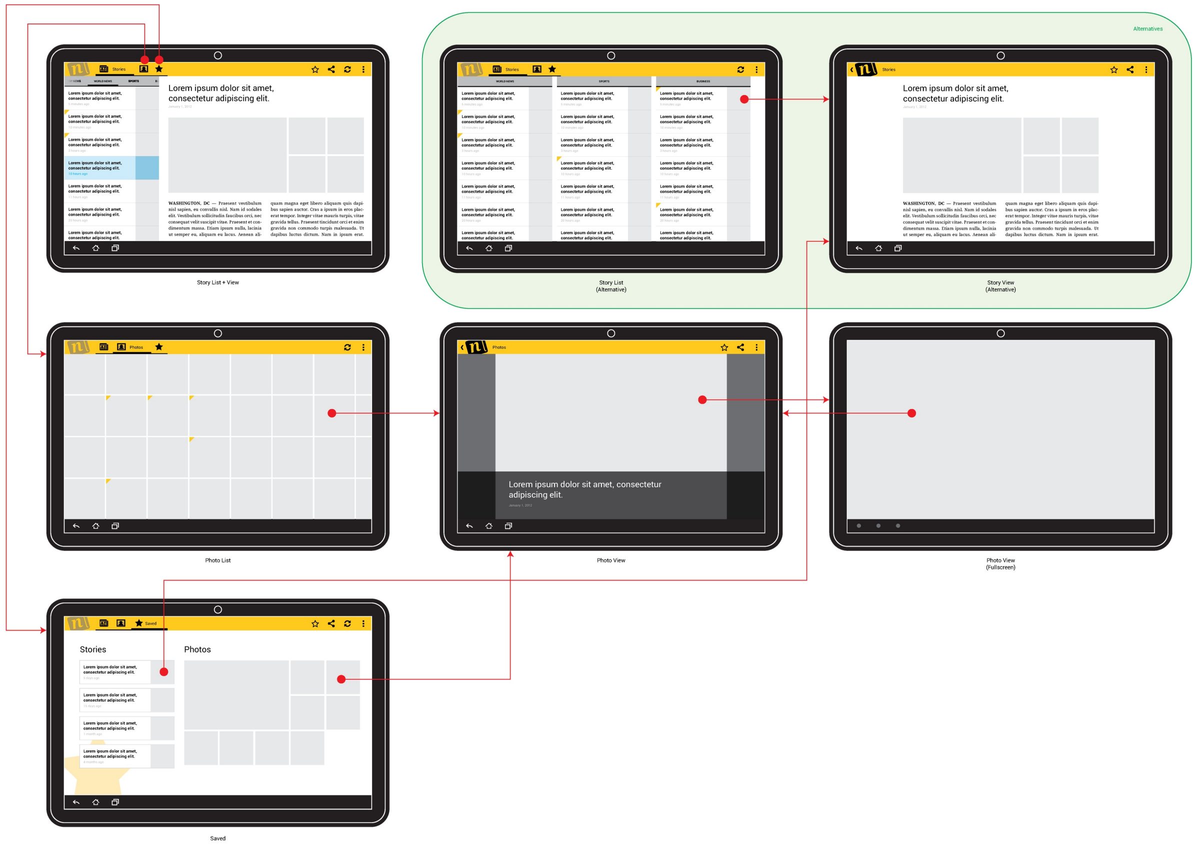


Figure 6. 新闻客户端平板横屏Wireframe样例（下载 SVG 图）

（下载表示设备的 Wireframe 的 SVG 图）

下一步

现在你已经为你的应用设计出了高效直观的 App 内部导航，你可用开始花时间来为单个界面改善 UI了。例如，展示交互内容时，你可以选择使用更花哨的控件来代替简单的文本标签，图像和按钮。你也可以开始定义你应用的视觉风格。在这过程中把你品牌的元素作为视觉语言融入其中吧。

最后，也适时实现你的设计吧，使用 Android SDK 为你的应用写写代码。想开始？看看下面的这些资源吧：

- 开发者指导：UI :学习如何用 Android SDK 实现你的 UI 设计。
- Action Bar :实现tab，向上导航，屏幕上动作，等等。
- Fragment :实现可重用，多视窗布局
- 支持库 :用 `ViewPager` 实现水平分页（Swipe View）

实现高效的导航

> 编写:Lin-H - 原文:http://developer.android.com/training/implementing-navigation/index.html

这节课将会演示如何实现在Designing Effective Navigation中所详述的关键导航设计模式。

在阅读这节课程内容之后，你会对如何使用tabs, swipe views, 和navigation drawer实现导航模式有一个深刻的理解。也会明白如何提供合适的向前向后导航(Up and Back navigation)。

> **Note**:本节课中的几个元素需要使用Support Library API。如果你之前没有使用过Support Library，可以按照Support Library Setup文档说明来使用。

Sample Code

EffectiveNavigation.zip

Lessons

- 使用Tabs创建Swipe View

  学习如何在action bar中实现tab，并提供横向分页(swipe views)在tab之间导航切换。

- 创建抽屉导航(Navigation Drawer)

  学习如何建立隐藏于屏幕边上的界面，通过划屏(swipe)或点击action bar中的app图标来显示这个界面。

- 提供向上导航

  学习如何使用action bar中的app图标实现向上导航

- 提供适当的向后导航

  学习如何正确处理特殊情况下的向后按钮(Back button)，包括在通知或app widget中的深度链接，如何将activity插入后退栈(back stack)中。

- 实现Descendant Navigation

  学习更精细地导航进入你的应用信息层。

使用Tabs创建Swipe视图

> 编写:Lin-H - 原文:http://developer.android.com/training/implementing-navigation/lateral.html

Swipe View提供在同级屏幕中的横向导航，例如通过横向划屏手势切换的tab(一种称作横向分页的模式)。这节课会教你如何使用swipe view创建一个tab layout实现在tab之间切换，或显示一个标题条替代tab。

> **Swipe View 设计**
>
> 在实现这些功能之前，你要先明白在Designing Effective Navigation, Swipe Views design guide中的概念和建议

实现Swipe View

你可以使用Support Library中的ViewPager控件在你的app中创建swipe view。ViewPager是一个子视图在layout上相互独立的布局控件(layout widget)。

使用ViewPager来设置你的layout，要添加一个 `<ViewPager>` 元素到你的XML layout中。例如，在你的swipe view中如果每一个页面都会占用整个layout，那么你的layout应该是这样:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

要插入每一个页面的子视图，你需要把这个layout与PagerAdapter挂钩。有两种adapter(适配器)你可以用:

FragmentPagerAdapter

在同级屏幕(sibling screen)只有少量的几个固定页面时，使用这个最好。

FragmentStatePagerAdapter

当根据对象集的数量来划分页面，即一开始页面的数量未确定时，使用这个最好。当用户切换到其他页面时，fragment会被销毁来降低内存消耗。

例如，这里的代码是当你使用FragmentStatePagerAdapter来在Fragment对象集合中进行横屏切换:

```
public class CollectionDemoActivity extends FragmentActivity {
    // 当被请求时, 这个adapter会返回一个DemoObjectFragment,
    // 代表在对象集中的一个对象.
    DemoCollectionPagerAdapter mDemoCollectionPagerAdapter;
    ViewPager mViewPager;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_collection_demo);

        // ViewPager和他的adapter使用了support library
        // fragments,所以要用getSupportFragmentManager.
        mDemoCollectionPagerAdapter =
                new DemoCollectionPagerAdapter(
                        getSupportFragmentManager());
        mViewPager = (ViewPager) findViewById(R.id.pager);
        mViewPager.setAdapter(mDemoCollectionPagerAdapter);
    }
}

// 因为这是一个对象集所以使用FragmentStatePagerAdapter,
// 而不是FragmentPagerAdapter.
public class DemoCollectionPagerAdapter extends FragmentStatePagerAdapter {
    public DemoCollectionPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public Fragment getItem(int i) {
        Fragment fragment = new DemoObjectFragment();
        Bundle args = new Bundle();
        // 我们的对象只是一个整数 :-P
        args.putInt(DemoObjectFragment.ARG_OBJECT, i + 1);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public int getCount() {
        return 100;
    }

    @Override
    public CharSequence getPageTitle(int position) {
        return "OBJECT " + (position + 1);
    }
}

// 这个类的实例是一个代表了数据集中一个对象的fragment
public static class DemoObjectFragment extends Fragment {
    public static final String ARG_OBJECT = "object";
```

```
    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        // 最后两个参数保证LayoutParam能被正确填充
        View rootView = inflater.inflate(
                R.layout.fragment_collection_object, container, false);
        Bundle args = getArguments();
        ((TextView) rootView.findViewById(android.R.id.text1)).setText(
                Integer.toString(args.getInt(ARG_OBJECT)));
        return rootView;
    }
}
```

这个例子只显示了创建swipe view的必要代码。下面一节向你说明如何通过添加tab使导航更方便在页面间切换。

添加Tab到Action Bar

Action bar tab能给用户提供更熟悉的界面来在app的同级屏幕中切换和分辨。

使用ActionBar来创建tab，你需要启用NAVIGATION_MODE_TABS，然后创建几个ActionBar.Tab的实例，并对每个实例实现ActionBar.TabListener接口。例如在你的activity的onCreate()方法中，你可以使用与下面相似的代码:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    final ActionBar actionBar = getActionBar();
    ...

    // 指定在action bar中显示tab.
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    // 创建一个tab listener，在用户切换tab时调用.
    ActionBar.TabListener tabListener = new ActionBar.TabListener() {
        public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
            // 显示指定的tab
        }

        public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction ft) {
            // 隐藏指定的tab
        }

        public void onTabReselected(ActionBar.Tab tab, FragmentTransaction ft) {
            // 可以忽略这个事件
        }
    };

    // 添加3个tab, 并指定tab的文字和TabListener
    for (int i = 0; i < 3; i++) {
        actionBar.addTab(
                actionBar.newTab()
                        .setText("Tab " + (i + 1))
                        .setTabListener(tabListener));
    }
}
```

根据你如何创建你的内容来处理ActionBar.TabListener回调改变tab。但是如果你是像上面那样，通过ViewPager对每个tab使用fragment，下面这节就会说明当用户选择一个tab时如何切换页面，当用户划屏切换页面时如何更新相应页面的tab。

使用Swipe View切换Tab

当用户选择tab时，在ViewPager中切换页面，需要实现ActionBar.TabListener来调用在ViewPager中的setCurrentItem()来选择相应的页面:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...

    // Create a tab listener that is called when the user changes tabs.
    ActionBar.TabListener tabListener = new ActionBar.TabListener() {
        public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
            // 当tab被选中时, 切换到ViewPager中相应的页面.
            mViewPager.setCurrentItem(tab.getPosition());
        }
        ...
    };
}
```

同样的，当用户通过触屏手势(touch gesture)切换页面时，你也应该选择相应的tab。你可以通过实现ViewPager.OnPageChangeListener接口来设置这个操作，当页面变化时当前的tab也相应变化。例如:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...

    mViewPager = (ViewPager) findViewById(R.id.pager);
    mViewPager.setOnPageChangeListener(
            new ViewPager.SimpleOnPageChangeListener() {
                @Override
                public void onPageSelected(int position) {
                    // 当划屏切换页面时，选择相应的tab.
                    getActionBar().setSelectedNavigationItem(position);
                }
            });
    ...
}
```

使用标题栏替代Tab

如果你不想使用action bar tab，而想使用scrollable tabs来提供一个更简短的可视化配置，你可以在swipe view中使用PagerTitleStrip。

下面是一个内容为ViewPager，有一个PagerTitleStrip顶端对齐的activity的layout XML文件示例。单个页面(adapter提供)占据ViewPager中的剩余空间。

```
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.view.PagerTitleStrip
        android:id="@+id/pager_title_strip"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:background="#33b5e5"
        android:textColor="#fff"
        android:paddingTop="4dp"
        android:paddingBottom="4dp" />

</android.support.v4.view.ViewPager>
```

创建抽屉式导航(navigation drawer)

> 编写:Lin-H - 原文: http://developer.android.com/training/implementing-navigation/nav-drawer.html

Navigation drawer是一个在屏幕左侧边缘显示导航选项的面板。大部分时候是隐藏的，当用户从屏幕左侧划屏，或在top level模式的app中点击action bar中的app图标时，才会显示。

这节课叙述如何使用Support Library中的DrawerLayout API，来实现navigation drawer。

> **Navigation Drawer 设计**：在你决定在你的app中使用Navigation Drawer之前，你应该先理解在Navigation Drawer design guide中定义的使用情况和设计准则。

创建一个Drawer Layout

要添加一个navigation drawer，在你的用户界面layout中声明一个用作root view(根视图)的DrawerLayout对象。在DrawerLayout中为屏幕添加一个包含主要内容的view(当drawer隐藏时的主layout)，和其他一些包含navigation drawer内容的view。

例如，下面的layout使用了有两个子视图(child view)的DrawerLayout:一个FrameLayout用来包含主要内容(在运行时被Fragment填入)，和一个navigation drawer使用的ListView。

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- 包含主要内容的 view -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <!-- navigation drawer(抽屉式导航) -->
    <ListView android:id="@+id/left_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>
```

这个layout展示了一些layout的重要特点:

- 主内容view(上面的FrameLayout)，在DrawerLayout中必须是第一个子视图，因为XML的顺序代表着Z轴(垂直于手机屏幕)的顺序，并且drawer必须在内容的前端。
- 主内容view被设置为匹配父视图的宽和高，因为当navigation drawer隐藏时，主内容表示整个UI部分。
- drawer视图(ListView)必须使用 `android:layout_gravity` 属性指定它的**horizontal gravity**。为了支持从右边阅读的语言(right-to-left(RTL) language)，指定它的值为 `"start"` 而不是 `"left"` (当layout是RTL时drawer在右边显示)。
- drawer视图以 `dp` 为单位指定它的宽和高来匹配父视图。drawer的宽度不能大于320dp，这样用户总能看到部分主内容。

初始化Drawer List

在你的activity中，首先要做的事就是要初始化drawer的item列表。这要根据你的app内容来处理，但是一个navigation drawer通常由一个ListView组成，所以列表应该通过一个Adapter(例如ArrayAdapter或SimpleCursorAdapter)填入。

例如，如何使用一个字符串数组(string array)来初始化导航列表(navigation list):

```
public class MainActivity extends Activity {
    private String[] mPlanetTitles;
    private DrawerLayout mDrawerLayout;
    private ListView mDrawerList;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mPlanetTitles = getResources().getStringArray(R.array.planets_array);
        mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
        mDrawerList = (ListView) findViewById(R.id.left_drawer);

        // 为list view设置adapter
        mDrawerList.setAdapter(new ArrayAdapter<String>(this,
                R.layout.drawer_list_item, mPlanetTitles));
        // 为list设置click listener
        mDrawerList.setOnItemClickListener(new DrawerItemClickListener());

        ...
    }
}
```

这段代码也调用了setOnItemClickListener()来接收navigation drawer列表的点击事件。下一节会说明如何实现这个接口，并且当用户选择一个item时如何改变内容视图(content view)。

处理导航的点击事件

当用户选择drawer列表中的item，系统会调用在setOnItemClickListener()中所设置的OnItemClickListener的onItemClick()。

在onItemClick()方法中做什么，取决于你如何实现你的app结构(app structure)。在下面的例子中，每选择一个列表中的item，就插入一个不同的Fragment到主内容视图中(FrameLayout元素通过 `R.id.content_frame` ID辨识):

```
private class DrawerItemClickListener implements ListView.OnItemClickListener {
    @Override
    public void onItemClick(AdapterView parent, View view, int position, long id) {
        selectItem(position);
    }
}

/** 在主内容视图中交换fragment */
private void selectItem(int position) {
    // 创建一个新的fragment并且根据行星的位置来显示
    Fragment fragment = new PlanetFragment();
    Bundle args = new Bundle();
    args.putInt(PlanetFragment.ARG_PLANET_NUMBER, position);
    fragment.setArguments(args);

    // 通过替换已存在的fragment来插入新的fragment
    FragmentManager fragmentManager = getFragmentManager();
    fragmentManager.beginTransaction()
                   .replace(R.id.content_frame, fragment)
                   .commit();

    // 高亮被选择的item, 更新标题, 并关闭drawer
    mDrawerList.setItemChecked(position, true);
    setTitle(mPlanetTitles[position]);
    mDrawerLayout.closeDrawer(mDrawerList);
}

@Override
public void setTitle(CharSequence title) {
    mTitle = title;
    getActionBar().setTitle(mTitle);
}
```

监听打开和关闭事件

要监听drawer的打开和关闭事件，在你的DrawerLayout中调用setDrawerListener()，并传入一个DrawerLayout.DrawerListener的实现。这个接口提供drawer事件的回调例如onDrawerOpened()和onDrawerClosed()。

但是，如果你的activity包含有action bar可以不用实现DrawerLayout.DrawerListener，你可以继承ActionBarDrawerToggle来替代。ActionBarDrawerToggle实现了DrawerLayout.DrawerListener，所以你仍然可以重写这些回调。这么做也能使action bar图标和 navigation drawer的交互操作变得更容易(在下节详述)。

如Navigation Drawer design guide中所述,当drawer可见时，你应该修改action bar的内容，比如改变标题和移除与主文字内容相关的action item。下面的代码向你说明如何通过ActionBarDrawerToggle类的实例，重写DrawerLayout.DrawerListener的回调方法来实现这个目的:

```java
public class MainActivity extends Activity {
    private DrawerLayout mDrawerLayout;
    private ActionBarDrawerToggle mDrawerToggle;
    private CharSequence mDrawerTitle;
    private CharSequence mTitle;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...

        mTitle = mDrawerTitle = getTitle();
        mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
        mDrawerToggle = new ActionBarDrawerToggle(this, mDrawerLayout,
                R.drawable.ic_drawer, R.string.drawer_open, R.string.drawer_close) {

            /** 当drawer处于完全关闭的状态时调用 */
            public void onDrawerClosed(View view) {
                super.onDrawerClosed(view);
                getActionBar().setTitle(mTitle);
                invalidateOptionsMenu(); // 创建对onPrepareOptionsMenu()的调用
            }

            /** 当drawer处于完全打开的状态时调用 */
            public void onDrawerOpened(View drawerView) {
                super.onDrawerOpened(drawerView);
                getActionBar().setTitle(mDrawerTitle);
                invalidateOptionsMenu(); // 创建对onPrepareOptionsMenu()的调用
            }
        };

        // 设置drawer触发器为DrawerListener
        mDrawerLayout.setDrawerListener(mDrawerToggle);
    }

    /* 当invalidateOptionsMenu()调用时调用 */
    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        // 如果nav drawer是打开的, 隐藏与内容视图相关联的action items
        boolean drawerOpen = mDrawerLayout.isDrawerOpen(mDrawerList);
        menu.findItem(R.id.action_websearch).setVisible(!drawerOpen);
        return super.onPrepareOptionsMenu(menu);
    }
}
```

下一节会描述ActionBarDrawerToggle的构造参数，和处理与action bar图标交互所需的其他步骤。

使用App图标来打开和关闭

用户可以在屏幕左侧使用划屏手势来打开和关闭navigation drawer，但是如果你使用action bar,你也应该允许用户通过点击app图标来打开或关闭。并且app图标也应该使用一个特殊的图标来指明navigation drawer的存在。你可以通过使用上一节所说的ActionBarDrawerToggle来实现所有的这些操作。

要使ActionBarDrawerToggle起作用，通过它的构造函数创建一个实例，需要用到以下参数:

- Activity用来容纳drawer。
- DrawerLayout。
- 一个drawable资源用作drawer指示器。 标准的navigation drawer可以在Download the Action Bar Icon Pack获的
- 一个字符串资源描述"打开抽屉"操作(便于访问)
- 一个字符串资源描述"关闭抽屉"操作(便于访问)

那么，不论你是否创建了用作drawer监听器的ActionBarDrawerToggle的子类，你都需要在activity生命周期中的某些地方根据你的ActionBarDrawerToggle来调用。

```
public class MainActivity extends Activity {
    private DrawerLayout mDrawerLayout;
    private ActionBarDrawerToggle mDrawerToggle;
    ...

    public void onCreate(Bundle savedInstanceState) {
        ...

        mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
        mDrawerToggle = new ActionBarDrawerToggle(
                this,                  /* 承载 Activity */
                mDrawerLayout,         /* DrawerLayout 对象 */
                R.drawable.ic_drawer,  /* nav drawer 图标用来替换'Up'符号 */
                R.string.drawer_open,  /* "打开 drawer" 描述 */
                R.string.drawer_close  /* "关闭 drawer" 描述 */
                ) {

            /** 当drawer处于完全关闭的状态时调用 */
            public void onDrawerClosed(View view) {
                super.onDrawerClosed(view);
                getActionBar().setTitle(mTitle);
            }

            /** 当drawer处于完全打开的状态时调用 */
            public void onDrawerOpened(View drawerView) {
                super.onDrawerOpened(drawerView);
                getActionBar().setTitle(mDrawerTitle);
```

```
            }
        };

        // 设置drawer触发器为DrawerListener
        mDrawerLayout.setDrawerListener(mDrawerToggle);

        getActionBar().setDisplayHomeAsUpEnabled(true);
        getActionBar().setHomeButtonEnabled(true);
    }

    @Override
    protected void onPostCreate(Bundle savedInstanceState) {
        super.onPostCreate(savedInstanceState);
        // 在onRestoreInstanceState发生后，同步触发器状态.
        mDrawerToggle.syncState();
    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        mDrawerToggle.onConfigurationChanged(newConfig);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // 将事件传递给ActionBarDrawerToggle, 如果返回true，表示app 图标点击事件已经被处理
        if (mDrawerToggle.onOptionsItemSelected(item)) {
          return true;
        }
        // 处理你的其他action bar items...

        return super.onOptionsItemSelected(item);
    }

    ...
}
```

一个完整的navigation drawer例子,可以在原文页面顶端的sample下载

提供向上的导航

> 编写:Lin-H - 原文:http://developer.android.com/training/implementing-navigation/ancestral.html

所有不是从主屏幕("home"屏幕)进入app的，都应该给用户提供一种方法，通过点击action bar中的Up按钮。可以回到app的结构层次中逻辑父屏幕。本课程向你说明如何正确地实现这一操作。

> **Up Navigation** 设计
>
> Designing Effective Navigation和the Navigation design guide中描述了向上导航的概念和设计准则。

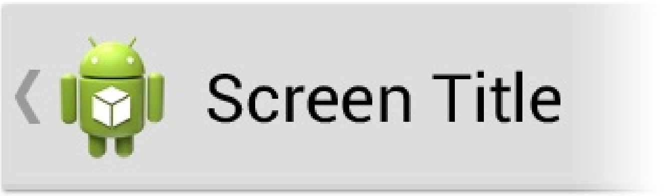


Figure 1. action bar中的Up按钮.

指定父Activity

要实现向上导航，第一步就是为每一个activity声明合适的父activity。这么做可以使系统简化导航模式，例如向上导航，因为系统可以从manifest文件中判断它的逻辑父(logical parent)activity。

从Android 4.1 (API level 16)开始，你可以通过指定 `<activity>` 元素中的android:parentActivityName属性来声明每一个activity的逻辑父activity。

如果你的app需要支持Android 4.0以下版本，在你的app中包含Support Library并添加 `<meta-data>` 元素到 `<activity>` 中。然后指定父activity的值为 `android.support.PARENT_ACTIVITY` ，并匹配android:parentActivityName的值。

例如:

```
<application ... >
    ...
    <!-- main/home activity (没有父activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- 主activity的一个子activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- 父activity的meta-data，用来支持4.0以下版本 -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

在父activity这样声明后，你可以使用NavUtils API进行向上导航操作，就像下一面这节。

添加向上操作(Up Action)

要使用action bar的app图标来完成向上导航，需要调用setDisplayHomeAsUpEnabled():

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

这样，在app旁添加了一个左向符号，并用作操作按钮。当用户点击它时，你的activity会接收一个对onOptionsItemSelected()的调用。操作的ID是 `android.R.id.home` 。

向上导航至父activity

要在用户点击app图标时向上导航，你可以使用NavUtils类中的静态方法navigateUpFromSameTask()。当你调用这一方法时，系统会结束当前的activity并启动(或恢复)相应的父activity。如果目标activity在任务的后退栈中(back stack)，则目标activity会像FLAG_ACTIVITY_CLEAR_TOP定义的那样，提到栈顶。提到栈顶的方式取决于父activity是否处理了对onNewIntent()的调用。

例如:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    // 对action bar的Up/Home按钮做出反应
    case android.R.id.home:
        NavUtils.navigateUpFromSameTask(this);
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

但是，只能是当你的**app**拥有当前任务**(current task)**(用户从你的app中发起这一任务)时navigateUpFromSameTask()才有用。如果你的activity是从别的app的任务中启动的话，向上导航操作就应该创建一个属于你的app的新任务，并需要你创建一个新的后退栈。

用新的后退栈来向上导航

如果你的activity提供了任何允许被别的app启动的intent filters，那么你应该实现onOptionsItemSelected()回调，在用户从别的app任务进入你的activity后，点击Up按钮，在向上导航之前你的app用相应的后退栈开启一个新的任务。

在这么做之前，你可以先调用shouldUpRecreateTask()来检查当前的activity实例是否在另一个不同的app任务中。如果返回true，就使用TaskStackBuilder创建一个新任务。或者，你可以向上面那样使用navigateUpFromSameTask()方法。

例如:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    // 对action bar的Up/Home按钮做出反应
    case android.R.id.home:
        Intent upIntent = NavUtils.getParentActivityIntent(this);
        if (NavUtils.shouldUpRecreateTask(this, upIntent)) {
            // 这个activity不是这个app任务的一部分, 所以当向上导航时创建
            // 用合成后退栈(synthesized back stack)创建一个新任务。
            TaskStackBuilder.create(this)
                    // 添加这个activity的所有父activity到后退栈中
                    .addNextIntentWithParentStack(upIntent)
                    // 向上导航到最近的一个父activity
                    .startActivities();
        } else {
            // 这个activity是这个app任务的一部分, 所以
            // 向上导航至逻辑父activity.
            NavUtils.navigateUpTo(this, upIntent);
        }
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

> **Note**:为了能使addNextIntentWithParentStack()方法起作用，你必须像上面说的那样，在你的manifest文件中使用android:parentActivityName(和相应的 `<meta-data>` 元素)属性声明所有的activity的逻辑父activity。

提供向后的导航

> 编写:Lin-H - 原文:http://developer.android.com/training/implementing-navigation/temporal.html

向后导航(Back navigation)是用户根据屏幕历史记录返回之前所查看的界面。所有Android设备都可以为这种导航提供后退按钮，所以你的**app**不需要在**UI**中添加后退按钮。

在几乎所有情况下，当用户在应用中进行导航时，系统会保存activity的后退栈。这样当用户点击后退按钮时，系统可以正确地向后导航。但是，有少数几种情况需要手动指定app的后退操作，来提供更好的用户体验。

> **Back Navigation** 设计
>
> 在继续阅读篇文章之前，你应该先在Navigation design guide中对后退导航的概念和设计准则有个了解。

手动指定后退操作需要的导航模式:

- 当用户从notification(通知)，app widget，navigation drawer直接进入深层次activity。
- 用户在fragment之间切换的某些情况。
- 当用户在WebView中对网页进行导航。

下面说明如何在这几种情况下实现恰当的向后导航。

为深度链接合并新的后退栈

一般而言，当用户从一个activity导航到下一个时，系统会递增地创建后退栈。但是当用户从一个在自己的任务中启动activity的深度链接进入app，你就有必要去同步新的后退栈，因为新的activity是运行在一个没有任何后退栈的任务中。

例如，当用户从通知进入你的app中的深层activity时，你应该添加别的activity到你的任务的后退栈中，这样当点击后退(Back)时向上导航，而不是退出app。这个模式在Navigation design guide中有更详细的介绍。

在manifest中指定父activity

从Android 4.1 (API level 16)开始，你可以通过指定 `<activity>` 元素中的android:parentActivityName属性来声明每一个activity的逻辑父activity。这样系统可以使导航模式变得更容易，因为系统可以根据这些信息判断逻辑Back Up navigation的路径。

如果你的app需要支持Android 4.0以下版本，在你的app中包含Support Library并添加 `<meta-data>` 元素到 `<activity>` 中。然后指定父activity的值为 `android.support.PARENT_ACTIVITY` ，并匹配android:parentActivityName的值。

例如:

```
<application ... >
    ...
    <!-- main/home activity (没有父activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- 主activity的一个子activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- 4.1 以下的版本需要使用meta-data元素 -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

当父activity用这种方式声明，你就可以使用NavUtils API，通过确定每个activity相应的父activity来同步新的后退栈。

在启动activity时创建后退栈

在发生用户进入app的事件时，开始添加activity到后退栈中。就是说，使用TaskStackBuilder API定义每个被放到新后退栈的activity，不使用startActivity()。然后调用startActivities()来启动目标activity，或调用getPendingIntent()来创建相应的PendingIntent。

例如，当用户从通知进入你的app中的深层activity时，你可以使用这段代码来创建一个启动activity并把新后退栈插入目标任务的PendingIntent。

```
// 当用户选择通知时，启动activity的intent
Intent detailsIntent = new Intent(this, DetailsActivity.class);

// 使用TaskStackBuilder创建后退栈，并获取PendingIntent
PendingIntent pendingIntent =
        TaskStackBuilder.create(this)
                        // 添加所有DetailsActivity的父activity到栈中,
                        // 然后再添加DetailsActivity自己
                        .addNextIntentWithParentStack(upIntent)
                        .getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);

NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
builder.setContentIntent(pendingIntent);
...
```

产生的PendingIntent不仅指定了启动哪个activity(被 `detailsIntent` 所定义)还指定了要插入任务(所有被 `detailsIntent` 定义的 `DetailsActivity`)的后退栈。所以当 `DetailsActivity` 启动时，点击Back向后导航至每一个 `DetailsActivity` 类的父activity。

> **Note**:为了使addNextIntentWithParentStack()方法起作用，像上面所说那样，你必须在你的manifest文件中使用android:parentActivityName(和相应的元素 `<meta-data>`)属性声明每个activity的逻辑父activity。

为Fragment实现向后导航

当在app中使用fragment时，个别的FragmentTransaction对象可以代表要加入后退栈中变化的内容。例如，如果你要在手机上通过交换fragment实现一个master/detail flow(主/详细流程)，你就要保证点击Back按钮可以从detail screen返回到master screen。要这么做，你可以在提交事务(transaction)之前调用addToBackStack():

```
// 使用framework FragmentManager
// 或support package FragmentManager (getSupportFragmentManager).
getSupportFragmentManager().beginTransaction()
                           .add(detailFragment, "detail")
                           // 提交这一事务到后退栈中
                           .addToBackStack()
                           .commit();
```

当后退栈中有FragmentTransaction对象并且用户点击Back按钮时,FragmentManager会从后退栈中弹出最近的事务，然后执行反向操作(例如如果事务添加了一个fragment，那么就删除一个fragment)。

> **Note**:当事务用作水平导航(例如切换tab)或者修改内容外观(例如在调整filter时)时，不要将这个事务添加到后退栈中。更多关于向后导航的恰当时间的信息，详见Navigation design guide。

如果你的应用更新了别的UI元素来反应当前的fragment状态，例如action bar，记得当你提交事务时更新UI。除了在提交事务的时候，在后退栈发生变化时也要更新你的UI。你可以设置一个FragmentManager.OnBackStackChangedListener来监听FragmentTransaction什么时候复原:

```
getSupportFragmentManager().addOnBackStackChangedListener(
        new FragmentManager.OnBackStackChangedListener() {
            public void onBackStackChanged() {
                // 在这里更新你的UI
            }
        });
```

为WebView实现向后导航

如果你的应用的一部分包含在WebView中，可以通过浏览器历史使用Back。要这么做，如果WebView有历史记录，你可以重写onBackPressed()并代理给WebView:

```
@Override
public void onBackPressed() {
    if (mWebView.canGoBack()) {
        mWebView.goBack();
        return;
    }

    // 否则遵从系统的默认操作.
    super.onBackPressed();
}
```

要注意当使用这一机制时，高动态化的页面会产生大量历史。会生成大量历史的页面，例如经常改变文件散列(document hash)的页面,当要退出你的activity时，这会使你的用户感到繁琐。

更多关于使用WebView的信息，详见Building Web Apps in WebView。

实现向下的导航

> 编写:Lin-H - 原文:http://developer.android.com/training/implementing-navigation/descendant.html

Descendant Navigation是用来向下导航至应用的信息层次。在Designing Effective Navigation和Android Design: Application Structure中说明。

Descendant navigation通常使用Intent和startActivity()实现，或使用FragmentTransaction对象添加fragment到一个activity中。这节课程涵盖了在实现Descendant navigation时遇到的其他有趣的情况。

在手机和平板(Tablet)上实现Master/Detail Flow

在master/detail导航流程(navigation flow)中，master screen(主屏幕)包含一个集合中item的列表，detail screen(详细屏幕)显示集合中特定item的详细信息。实现从master screen到detail screen的导航是Descendant Navigation的一种形式。

手机触摸屏非常适合一次显示一种屏幕(master screen或detail screen)；这一想法在Planning for Multiple Touchscreen Sizes中进一步说明。在这种情况下，一般使用Intent启动detail screen来实现activityDescendant navigation。另一方面，平板的显示，特别是用横屏来浏览时，最适合一次显示多个内容窗格，master内容在左边，detail在右边。在这里一般就使用FragmentTransaction实现descendant navigation。FragmentTransaction用来添加、删除或用新内容替换detail窗格(pane)。

实现这一模式的基础内容在Designing for Multiple Screens的Implementing Adaptive UI Flows课程中说明。课程中说明了如何在手机上使用两个activity，在平板上使用一个activity来实现master/detail flow。

导航至外部Activities

有很多情况，是从别的应用下降(descend)至你的应用信息层次(application's information hierarchy)再到activity。例如，当正在浏览手机通讯录中联系信息的details screen，子屏幕详细显示由社交网络联系提供的最近文章，子屏幕可就可以属于一个社交网络应用。

当启动另一个应用的activity来允许用户说话，发邮件或选择一个照片附件，如果用户是从启动器(设备的home屏幕)重启你的应用，你一般不会希望用户返回到别的activity。如果点击你的应用图标又回到“发邮件”的屏幕，这会使用户感到很迷惑。

为防止这种情况的发生，只需要添加FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET标记到用来启动外部activity的intent中，就像:

```
Intent externalActivityIntent = new Intent(Intent.ACTION_PICK);
externalActivityIntent.setType("image/*");
externalActivityIntent.addFlags(
        Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
startActivity(externalActivityIntent);
```

通知提示用户

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/index.html

- Notification是一种在你APP常规UI外展示、用来指示某个事件发生的用户交互元素。用户可以在使用其它apps时查看notification，并在方便的时候做出回应。
- Notification设计指导向你展示如何设计实用的notifications以及何时使用它们。这节课将会教你实现大多数常用的notification设计。
- 完整的Demo示例：NotifyUser.zip

Lessons

- 建立一个Notification

  学习如何创建一个notification Builder，设置需要的特征，以及发布notification。

- 当Activity启动时保留导航

  学习如何为一个从notification启动的Activity执行适当的导航。

- 更新notifications

  学习如何更新与移除notifications

- 使用BigView风格

  学习用扩展的notification来创建一个BigView，并且维持老版本的兼容性。

- 显示notification进度

  学习在notification中显示某个操作的进度，既可以用于那些你可以估算已经完成多少（确定进度，determinate）的操作，也可以用于那些你无法知道完成了多少（不确定进度，indefinite ）的操作

建立一个Notification

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/build-notification.html

- 这节课向你说明如何创建与发布一个Notification。
- 这节课的例子是基于NotificationCompat.Builder类的，NotificationCompat.Builder在Support Library中。为了给许多各种不同的平台提供最好的notification支持，你应该使用NotificationCompat以及它的子类，特别是NotificationCompat.Builder。

创建Notification Buider

- 创建Notification时，可以用NotificationCompat.Builder对象指定Notification的UI内容与行为。一个Builder至少包含以下内容：
  - 一个小的icon，用setSmallIcon())方法设置
  - 一个标题，用setContentTitle())方法设置。
  - 详细的文本，用setContentText())方法设置

例如：

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("My notification")
    .setContentText("Hello World!");
```

定义Notification的Action（行为）

- 尽管在Notification中Actions是可选的，但是你应该至少添加一种Action。一种Action可以让用户从Notification直接进入你应用内的Activity，在这个activity中他们可以查看引起Notification的事件或者做下一步的处理。在Notification中，action本身是由PendingIntent定义的，PendingIntent包含了一个启动你应用内Activity的Intent。
- 如何构建一个PendingIntent取决于你要启动的activity的类型。当从Notification中启动一个activity时，你必须保存用户的导航体验。在下面的代码片段中，点击Notification启动一个新的activity，这个activity有效地扩展了Notification的行为。在这种情形下，就没必

> 要人为地去创建一个返回栈（更多关于这方面的信息，请查看 Preserving Navigation when Starting an Activity）

```
Intent resultIntent = new Intent(this, ResultActivity.class);
...
// Because clicking the notification opens a new ("special") activity, there's
// no need to create an artificial back stack.
PendingIntent resultPendingIntent =
    PendingIntent.getActivity(
    this,
    0,
    resultIntent,
    PendingIntent.FLAG_UPDATE_CURRENT
);
```

设置Notification的点击行为

可以通过调用NotificationCompat.Builder中合适的方法，将上一步创建的PendingIntent与一个手势产生关联。比方说，当点击Notification抽屉里的Notification文本时，启动一个activity，可以通过调用setContentIntent())方法把PendingIntent添加进去。

例如：

```
PendingIntent resultPendingIntent;
...
mBuilder.setContentIntent(resultPendingIntent);
```

发布Notification

为了发布notification：

```
* 获取一个[NotificationManager](http://www.baidu.com/baidu?wd=NotificationManager.&tn=monline_4_dg)实例
* 使用[notify()](developer.android.com/reference/java/lang/Object.html#notify())方法发布Notification。当你调用[notify()](developer.android.com/reference/java/lang/Object.html#notify())方法时，指定一个notification ID。你可以在以后使用这个ID来更新你的notification。这在[Managing Notifications](developer.android.com/intl/zh-cn/training/notify-user/managing.html)中有更详细的描述。
* 调用[build()](developer.android.com/reference/android/support/v4/app/NotificationCompat.Builder.html#build())方法，会返回一个包含你的特征的[Notification](developer.android.com/reference/android/app/Notification.html)对象。
```

举个例子：

```
NotificationCompat.Builder mBuilder;
...
// Sets an ID for the notification
int mNotificationId = 001;
// Gets an instance of the NotificationManager service
NotificationManager mNotifyMgr =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
// Builds the notification and issues it.
mNotifyMgr.notify(mNotificationId, mBuilder.build());
```

启动Activity时保留导航

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/navigation.html

部分设计一个notification的目的是为了保持用户的导航体验。为了详细讨论这个课题，请看Notifications API引导，分为下列两种主要情况：

```
* 常规的activity
你启动的是你application工作流中的一部分[Activity](developer.android.com/reference/android/app/Activity.html)。
* 特定的activity
用户只能从notification中启动，才能看到这个[Activity](http://developer.android.com/intl/zh-cn/reference/android/app/Activity.html)，在某种意义上，这个[Activity](http://developer.android.com/intl/zh-cn/reference/android/app/Activity.html)是notification的扩展，额外展示了一些notification本身难以展示的信息。
```

设置一个常规的Activity PendingIntent

设置一个直接启动的入口Activity的PendingIntent，遵循以下步骤：

1 在manifest中定义你application的Activity层次，最终的manifest文件应该像这个：

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

2 在基于启动Activity的Intent中创建一个返回栈，比如：

```
int id = 1;
...
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
// Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
        stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
...
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
builder.setContentIntent(resultPendingIntent);
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(id, builder.build());
```

设置一个特定的Activity PendingIntent

一个特定的Activity不需要一个返回栈，所以你不需要在manifest中定义Activity的层次，以及你不需要调用 addParentStack())方法去构建一个返回栈。作为代替，你需要用manifest设置Activity任务选项，以及调用 getActivity())创建PendingIntent

1. manifest中，在Activity的 标签中增加下列属性： android:name="activityclass" activity的完整的类名。 android:taskAffinity="" 结合你在代码里设置的FLAG_ACTIVITY_NEW_TASK标识， 确保这个Activity不会进入application的默认任务。任何与 application的默认任务有密切关系的任务都不会受到影响。 android:excludeFromRecents="true" 将新任务从最近列表中排除，目的是为了防止用户不小心返回到它。

2. 建立以及发布notification： a.创建一个启动Activity的Intent. b.通过调用setFlags())方法并设置标识FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_CLEAR_TASK，来设置Activity在一个新的，空的任务中启动。 c.在Intent中设置其他你需要的选项。 d.通过调用 getActivity()方法从Intent中创建一个 PendingIntent，你可以把这个PendingIntent 当做 setContentIntent()的参数来使用。 下面的代码片段演示了这个过程：

```
// Instantiate a Builder object.
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
// Creates an Intent for the Activity
Intent notifyIntent =
        new Intent(new ComponentName(this, ResultActivity.class));
// Sets the Activity to start in a new, empty task
notifyIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TASK);
// Creates the PendingIntent
PendingIntent notifyIntent =
        PendingIntent.getActivity(
        this,
        0,
        notifyIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
);

// Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
// Notifications are issued by sending them to the
// NotificationManager system service.
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Builds an anonymous Notification object from the builder, and
// passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());
```

更新Notification

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/managing.html

当你需要对同一事件发布多次Notification时，你应该避免每次都生成一个全新的Notification。相反，你应该考虑去更新先前的Notification，或者改变它的值，或者增加一些值，或者两者同时进行。

下面的章节描述了如何更新Notifications，以及如何移除它们。

改变一个Notification

想要设置一个可以被更新的Notification，需要在发布它的时候调用NotificationManager.notify(ID, notification))方法为它指定一个notification ID。更新一个已经发布的Notification，需要更新或者创建一个NotificationCompat.Builder对象，并从这个对象创建一个Notification对象，然后用与先前一样的ID去发布这个Notification。

下面的代码片段演示了更新一个notification来反映事件发生的次数，它把notification堆积起来，显示一个总数。

```
mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Sets an ID for the notification, so it can be updated
int notifyID = 1;
mNotifyBuilder = new NotificationCompat.Builder(this)
    .setContentTitle("New Message")
    .setContentText("You've received new messages.")
    .setSmallIcon(R.drawable.ic_notify_status)
numMessages = 0;
// Start of a loop that processes data and then notifies the user
...
    mNotifyBuilder.setContentText(currentText)
        .setNumber(++numMessages);
    // Because the ID remains unchanged, the existing notification is
    // updated.
    mNotificationManager.notify(
            notifyID,
            mNotifyBuilder.build());
...
```

移除Notification

Notifications 将持续可见，除非下面任何一种情况发生。

```
* 用户清除Notification单独地或者使用“清除所有”（如果Notification能被清除）。
* 你在创建notification时调用了 setAutoCancel(developer.android.com/reference/android/supp
ort/v4/app/NotificationCompat.Builder.html#setAutoCancel(boolean))方法，以及用户点击了这个
notification，
* 你为一个指定的 notification ID调用了[cancel()](developer.android.com/reference/android/
app/NotificationManager.html#cancel(int))方法。这个方法也会删除正在进行的notifications。
* 你调用了[cancelAll()](developer.android.com/reference/android/app/NotificationManager.
html#cancelAll())方法，它将会移除你先前发布的所有Notification。
```

使用BigView样式

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/expanded.html

Notification抽屉中的Notification主要有两种视觉展示形式，normal view（平常的视图，下同） 与 big view（大视图，下同）。Notification的 big view样式只有当Notification被扩展时才能出现。当Notification在Notification抽屉的最上方或者用户点击Notification时才会展现大视图。

Big views在Android4.1被引进的，它不支持老版本设备。这节课叫你如何让把big view notifications合并进你的APP，同时提供normal view的全部功能。更多信息请见Notifications API guide 。

这是一个 normal view的例子

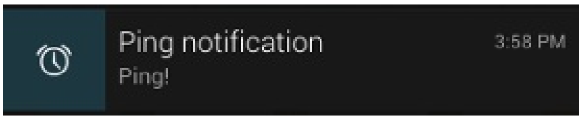


图1 Normal view notification.

这是一个 big view的例子

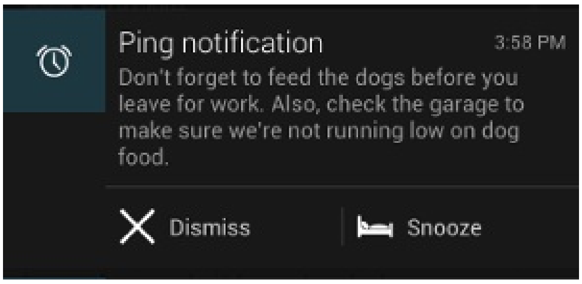


图2 Big view notification.

在这节课的例子应用中， normal view 与 big view给用户相同的功能：

- 继续小睡或者消除Notification
- 一个查看用户设置的类似计时器的提醒文字的方法，

- normal view 通过当用户点击Notification来启动一个新的activity的方式提供这些特性，记住当你设计你的notifications时，首先在normal view 中提供这些功能，因为很多用户会与notification交互。

设置Notification用来登陆一个新的Activity

这个例子应用用IntentService的子类（PingService）来构造以及发布notification。 在这个代码片段中，IntentService中的方法onHandleIntent()) 指定了当用户点击notification时启动一个新的activity。方法setContentIntent())定义了pending intent在用户点击notification时被激发，因此登陆这个activity.

```
Intent resultIntent = new Intent(this, ResultActivity.class);
resultIntent.putExtra(CommonConstants.EXTRA_MESSAGE, msg);
resultIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TASK);

// Because clicking the notification launches a new ("special") activity,
// there's no need to create an artificial back stack.
PendingIntent resultPendingIntent =
         PendingIntent.getActivity(
         this,
         0,
         resultIntent,
         PendingIntent.FLAG_UPDATE_CURRENT
);

// This sets the pending intent that should be fired when the user clicks the
// notification. Clicking the notification launches a new activity.
builder.setContentIntent(resultPendingIntent);
```

构造big view

这个代码片段展示了如何在big view中设置buttons

```
// Sets up the Snooze and Dismiss action buttons that will appear in the
// big view of the notification.
Intent dismissIntent = new Intent(this, PingService.class);
dismissIntent.setAction(CommonConstants.ACTION_DISMISS);
PendingIntent piDismiss = PendingIntent.getService(this, 0, dismissIntent, 0);

Intent snoozeIntent = new Intent(this, PingService.class);
snoozeIntent.setAction(CommonConstants.ACTION_SNOOZE);
PendingIntent piSnooze = PendingIntent.getService(this, 0, snoozeIntent, 0);
```

这个代码片段展示了如何构造一个Builder对象，它设置了big view 的样式为"big text",同时设置了它的内容为提醒文字。它使用addAction())方法来添加将要在big view中出现的Snooze与Dismiss按钮（以及它们相关联的pending intents).

```
// Constructs the Builder object.
NotificationCompat.Builder builder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_stat_notification)
        .setContentTitle(getString(R.string.notification))
        .setContentText(getString(R.string.ping))
        .setDefaults(Notification.DEFAULT_ALL) // requires VIBRATE permission
        /*
         * Sets the big view "big text" style and supplies the
         * text (the user's reminder message) that will be displayed
         * in the detail area of the expanded notification.
         * These calls are ignored by the support library for
         * pre-4.1 devices.
         */
        .setStyle(new NotificationCompat.BigTextStyle()
                .bigText(msg))
        .addAction (R.drawable.ic_stat_dismiss,
                getString(R.string.dismiss), piDismiss)
        .addAction (R.drawable.ic_stat_snooze,
                getString(R.string.snooze), piSnooze);
```

显示Notification进度

> 编写:fastcome1985 - 原文:http://developer.android.com/training/notify-user/display-progress.html

Notifications可以包含一个展示用户正在进行的操作状态的动画进度指示器。如果你可以在任何时候估算这个操作得花多少时间以及当前已经完成多少，你可以用“determinate（确定的，下同）”形式的指示器（一个进度条）。如果你不能估算这个操作的长度，使用“indeterminate（不确定，下同）”形式的指示器（一个活动的指示器）。

进度指示器用ProgressBar平台实现类来显示。

使用进度指示器，可以调用 setProgress()方法。determinate 与 indeterminate形式将在下面的章节中介绍。

展示固定长度的进度指示器

为了展示一个确定长度的进度条，调用 setProgress(max, progress, false))方法将进度条添加进notification，然后发布这个notification，第三个参数是个boolean类型，决定进度条是indeterminate (true) 还是 determinate (false)。在你操作进行时，增加progress，更新notification。在操作结束时，progress应该等于max。一个常用的调用 setProgress())的方法是设置max为100，然后增加progress就像操作的“完成百分比”。

当操作完成的时候，你可以选择或者让进度条继续展示，或者移除它。无论哪种情况下，记得更新notification的文字来显示操作完成。移除进度条，调用setProgress(0, 0, false))方法.比如：

```
int id = 1;
...
mNotifyManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mBuilder = new NotificationCompat.Builder(this);
mBuilder.setContentTitle("Picture Download")
    .setContentText("Download in progress")
    .setSmallIcon(R.drawable.ic_notification);
// Start a lengthy operation in a background thread
new Thread(
    new Runnable() {
        @Override
        public void run() {
            int incr;
            // Do the "lengthy" operation 20 times
            for (incr = 0; incr <= 100; incr+=5) {
                    // Sets the progress indicator to a max value, the
                    // current completion percentage, and "determinate"
                    // state
                    mBuilder.setProgress(100, incr, false);
                    // Displays the progress bar for the first time.
                    mNotifyManager.notify(id, mBuilder.build());
                        // Sleeps the thread, simulating an operation
                        // that takes time
                        try {
                            // Sleep for 5 seconds
                            Thread.sleep(5*1000);
                        } catch (InterruptedException e) {
                            Log.d(TAG, "sleep failure");
                        }
            }
            // When the loop is finished, updates the notification
            mBuilder.setContentText("Download complete")
            // Removes the progress bar
                    .setProgress(0,0,false);
            mNotifyManager.notify(id, mBuilder.build());
        }
    }
// Starts the thread by calling the run() method in its Runnable
).start();
```

结果notifications显示在图1中，左边是操作正在进行中的notification的快照，右边是操作已经完成的notification的快照。

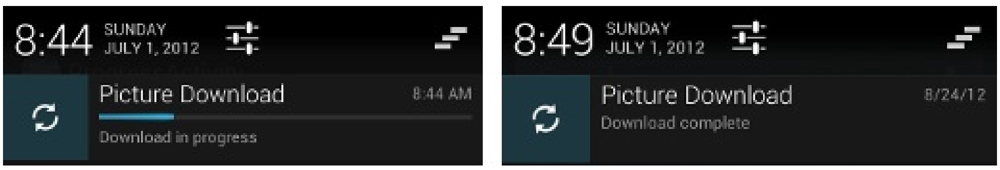


图1 操作正在进行中与完成时的进度条

展示持续的活动的指示器

为了展示一个持续的(indeterminate)活动的指示器,用setProgress(0, 0, true))方法把指示器添加进notification，然后发布这个notification 。前两个参数忽略，第三个参数决定indicator 还是indeterminate。结果是指示器与进度条有同样的样式，除了它的动画正在进行。

在操作开始的时候发布notification，动画将会一直进行直到你更新notification。当操作完成时，调用 setProgress(0, 0, false)) 方法，然后更新notification来移除这个动画指示器。一定要这么做，否责即使你操作完成了，动画还是会在那运行。同时也要记得更新notification的文字来显示操作完成。

为了观察持续的活动的指示器是如何工作的，看前面的代码。定位到下面的几行：

```
// Sets the progress indicator to a max value, the current completion
// percentage, and "determinate" state
mBuilder.setProgress(100, incr, false);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

将你找到的代码用下面的几行代码代替，注意 setProgress()方法的第三个参数设置成了true,表示进度条是 indeterminate类型的。

```
// Sets an activity indicator for an operation of indeterminate length
mBuilder.setProgress(0, 0, true);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

结果显示在图2中:

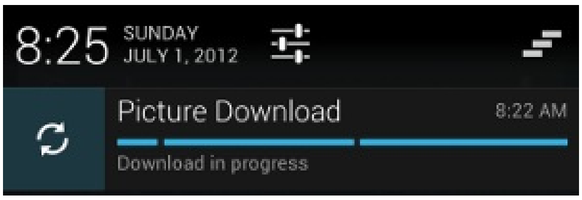


图2 正在进行的活动的指示器

增加搜索功能

> 编写:Lin-H - 原文:http://developer.android.com/training/search/index.html

Android的内置搜索功能，能够在app中方便地为所有用户提供一个统一的搜索体验。根据设备所运行的Android版本，有两种方式可以在你的app中实现搜索。本节课程涵盖如何像Android 3.0中介绍的那样用SearchView添加搜索，使用系统提供的默认搜索框来向下兼容旧版本Android。

Lessons

- 建立搜索界面

  学习如何向你的app中添加搜索界面，如何设置activity去处理搜索请求

- 保存并搜索数据

  学习在SQLite虚拟数据库表中用简单的方法储存和搜索数据

- 保持向下兼容

  通过使用搜索功能来学习如何向下兼容旧版本设备

建立搜索界面

> 编写:Lin-H - 原文:http://developer.android.com/training/search/setup.html

从Android 3.0开始，在action bar中使用SearchView作为item，是在你的app中提供搜索的一种更好方法。像其他所有在action bar中的item一样，你可以定义SearchView在有足够空间的时候总是显示，或设置为一个折叠操作(collapsible action),一开始SearchView作为一个图标显示，当用户点击图标时再显示搜索框占据整个action bar。

> **Note**:在本课程的后面，你会学习对那些不支持SearchView的设备，如何使你的app向下兼容至Android 2.1(API level 7)版本。

添加Search View到action bar中

为了在action bar中添加SearchView，在你的工程目录 `res/menu/` 中创建一个名为 `options_menu.xml` 的文件，再把下列代码添加到文件中。这段代码定义了如何创建search item，比如使用的图标和item的标题。 `collapseActionView` 属性允许你的SearchView占据整个action bar，在不使用的时候折叠成普通的action bar item。由于在手持设备中action bar的空间有限，建议使用 `collapsibleActionView` 属性来提供更好的用户体验。

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/search"
          android:title="@string/search_title"
          android:icon="@drawable/ic_search"
          android:showAsAction="collapseActionView|ifRoom"
          android:actionViewClass="android.widget.SearchView" />
</menu>
```

> **Note**:如果你的menu items已经有一个XML文件，你可以只把 `<item>` 元素添加入文件。

要在action bar中显示SearchView，在你的activity中onCreateOptionsMenu()方法内填充XML菜单资源(`res/menu/options_menu.xml`):

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    return true;
}
```

如果你立即运行你的app，SearchView就会显示在你app的action bar中，但还无法使用。你现在需要定义SearchView如何运行。

创建一个检索配置

检索配置(searchable configuration)在 `res/xml/searchable.xml` 文件中定义了SearchView如何运行。检索配置中至少要包含一个 `android:label` 属性，与Android manifest中的 `<application>` 或 `<activity>` `android:label` 属性值相同。但我们还是建议添加 `android:hint` 属性来告诉用户应该在搜索框中输入什么内容:

```
<?xml version="1.0" encoding="utf-8"?>

<searchable xmlns:android="http://schemas.android.com/apk/res/android"
        android:label="@string/app_name"
        android:hint="@string/search_hint" />
```

在你的应用的manifest文件中，声明一个指向 `res/xml/searchable.xml` 文件的 `<meta-data>` 元素，来告诉你的应用在哪里能找到检索配置。在你想要显示SearchView的 `<activity>` 中声明 `<meta-data>` 元素:

```
<activity ... >
    ...
    <meta-data android:name="android.app.searchable"
            android:resource="@xml/searchable" />

</activity>
```

在你之前创建的onCreateOptionsMenu())方法中，调用setSearchableInfo(SearchableInfo))把SearchView和检索配置关联在一起:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    // 关联检索配置和SearchView
    SearchManager searchManager =
           (SearchManager) getSystemService(Context.SEARCH_SERVICE);
    SearchView searchView =
            (SearchView) menu.findItem(R.id.search).getActionView();
    searchView.setSearchableInfo(
            searchManager.getSearchableInfo(getComponentName()));

    return true;
}
```

调用getSearchableInfo())返回一个SearchableInfo由检索配置XML文件创建的对象。检索配置与SearchView正确关联后，当用户提交一个搜索请求时，SearchView会以ACTION_SEARCH intent启动一个activity。所以你现在需要一个能过滤这个intent和处理搜索请求的activity。

创建一个检索activity

当用户提交一个搜索请求时，SearchView会尝试以ACTION_SEARCH启动一个activity。检索activity会过滤ACTION_SEARCH intent并在某种数据集中根据请求进行搜索。要创建一个检索activity，在你选择的activity中声明对ACTION_SEARCH intent过滤:

```
<activity android:name=".SearchResultsActivity" ... >
    ...
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    ...
</activity>
```

在你的检索activity中，通过在onCreate())方法中检查ACTION_SEARCH intent来处理它。

> **Note**:如果你的检索activity在single top mode下启动(`android:launchMode="singleTop"`)，也要在onNewIntent())方法中处理ACTION_SEARCH intent。在single top mode下你的activity只有一个会被创建，而随后启动的activity将不会在栈中创建新的activity。这种启动模式很有用，因为用户可以在当前activity中进行搜索，而不用在每次搜索时都创建一个activity实例。

```
public class SearchResultsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        handleIntent(getIntent());
    }

    @Override
    protected void onNewIntent(Intent intent) {
        ...
        handleIntent(intent);
    }

    private void handleIntent(Intent intent) {

        if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
            String query = intent.getStringExtra(SearchManager.QUERY);
            //通过某种方法，根据请求检索你的数据
        }
    }
    ...
}
```

如果你现在运行你的app，SearchView就能接收用户的搜索请求，以ACTION_SEARCH intent启动你的检索activity。现在就由你来解决如何依据请求来储存和搜索数据。

保存并搜索数据

> 编写:Lin-H - 原文:http://developer.android.com/training/search/search.html

有很多方法可以储存你的数据，比如储存在线上的数据库，本地的SQLite数据库，甚至是文本文件。你自己来选择最适合你应用的存储方式。本节课程会向你展示如何创建一个健壮的可以提供全文搜索的SQLite虚拟表。并从一个每行有一组单词-解释对的文件中将数据填入。

创建虚拟表

虚拟表与SQLite表的运行方式类似，但虚拟表是通过回调来向内存中的对象进行读取和写入，而不是通过数据库文件。要创建一个虚拟表，首先为该表创建一个类:

```
public class DatabaseTable {
    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }
}
```

在 `DatabaseTable` 类中创建一个继承SQLiteOpenHelper的内部类。你必须重写类SQLiteOpenHelper中定义的abstract方法，才能在必要的时候创建和更新你的数据库表。例如，下面一段代码声明了一个数据库表，用来储存字典app所需的单词。

```java
public class DatabaseTable {

    private static final String TAG = "DictionaryDatabase";

    //字典的表中将要包含的列项
    public static final String COL_WORD = "WORD";
    public static final String COL_DEFINITION = "DEFINITION";

    private static final String DATABASE_NAME = "DICTIONARY";
    private static final String FTS_VIRTUAL_TABLE = "FTS";
    private static final int DATABASE_VERSION = 1;

    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }

    private static class DatabaseOpenHelper extends SQLiteOpenHelper {

        private final Context mHelperContext;
        private SQLiteDatabase mDatabase;

        private static final String FTS_TABLE_CREATE =
                    "CREATE VIRTUAL TABLE " + FTS_VIRTUAL_TABLE +
                    " USING fts3 (" +
                    COL_WORD + ", " +
                    COL_DEFINITION + ")";

        DatabaseOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
            mHelperContext = context;
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            mDatabase = db;
            mDatabase.execSQL(FTS_TABLE_CREATE);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                    + newVersion + ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS " + FTS_VIRTUAL_TABLE);
            onCreate(db);
        }
    }
}
```

填入虚拟表

现在，表需要数据来储存。下面的代码会向你展示如何读取一个内容为单词和解释的文本文件(位于 `res/raw/definitions.txt`)，如何解析文件与如何将文件中的数据按行插入虚拟表中。为防止UI锁死这些操作会在另一条线程中执行。将下面的一段代码添加到你的 `DatabaseOpenHelper` 内部类中。

> Tip:你也可以设置一个回调来通知你的UI activity线程的完成结果。

```
private void loadDictionary() {
        new Thread(new Runnable() {
            public void run() {
                try {
                    loadWords();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        }).start();
    }

private void loadWords() throws IOException {
    final Resources resources = mHelperContext.getResources();
    InputStream inputStream = resources.openRawResource(R.raw.definitions);
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));

    try {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] strings = TextUtils.split(line, "-");
            if (strings.length < 2) continue;
            long id = addWord(strings[0].trim(), strings[1].trim());
            if (id < 0) {
                Log.e(TAG, "unable to add word: " + strings[0].trim());
            }
        }
    } finally {
        reader.close();
    }
}

public long addWord(String word, String definition) {
    ContentValues initialValues = new ContentValues();
    initialValues.put(COL_WORD, word);
    initialValues.put(COL_DEFINITION, definition);

    return mDatabase.insert(FTS_VIRTUAL_TABLE, null, initialValues);
}
```

任何恰当的地方，都可以调用 `loadDictionary()` 方法向表中填入数据。一个比较好的地方是 `DatabaseOpenHelper` 类的onCreate())方法中，紧随创建表之后:

```
@Override
public void onCreate(SQLiteDatabase db) {
    mDatabase = db;
    mDatabase.execSQL(FTS_TABLE_CREATE);
    loadDictionary();
}
```

搜索请求

当你的虚拟表创建好并填入数据后，根据SearchView提供的请求搜索数据。将下面的方法添加到 `DatabaseTable` 类中，用来创建搜索请求的SQL语句:

```
public Cursor getWordMatches(String query, String[] columns) {
    String selection = COL_WORD + " MATCH ?";
    String[] selectionArgs = new String[] {query+"*"};

    return query(selection, selectionArgs, columns);
}

private Cursor query(String selection, String[] selectionArgs, String[] columns) {
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(FTS_VIRTUAL_TABLE);

    Cursor cursor = builder.query(mDatabaseOpenHelper.getReadableDatabase(),
            columns, selection, selectionArgs, null, null, null);

    if (cursor == null) {
        return null;
    } else if (!cursor.moveToFirst()) {
        cursor.close();
        return null;
    }
    return cursor;
}
```

调用 `getWordMatches()` 来搜索请求。任何符合的结果返回到Cursor中，可以直接遍历或是建立一个ListView。这个例子是在检索activity的 `handleIntent()` 方法中调用 `getWordMatches()` 。请记住，因为之前创建的intent filter，检索activity会在ACTION_SEARCH intent中额外接收请求作为变量存储:

```
DatabaseTable db = new DatabaseTable(this);

...

private void handleIntent(Intent intent) {

    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        Cursor c = db.getWordMatches(query, null);
        //执行Cursor并显示结果
    }
}
```

保持向下兼容

> 编写:Lin-H - 原文:http://developer.android.com/training/search/backward-compat.html

SearchView和action bar只在Android 3.0以及以上版本可用。为了支持旧版本平台，你可以回到搜索对话框。搜索框是系统提供的UI，在调用时会覆盖在你的应用的最顶端。

设置最小和目标API级别

要设置搜索对话框，首先在你的manifest中声明你要支持旧版本设备，并且目标平台为Android 3.0或更新版本。当你这么做之后，你的应用会自动地在Android 3.0或以上使用action bar，在旧版本的设备使用传统的目录系统:

```
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="15" />

<application>
...
```

为旧版本设备提供搜索对话框

要在旧版本设备中调用搜索对话框，可以在任何时候，当用户从选项目录中选择搜索项时，调用onSearchRequested()。因为Android 3.0或以上会在action bar中显示SearchView(就像在第一节课中演示的那样)，所以当用户选择目录的搜索项时，只有Android 3.0以下版本的会调用onOptionsItemSelected()。

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.search:
            onSearchRequested();
            return true;
        default:
            return false;
    }
}
```

在运行时检查Android的构建版本

在运行时，检查设备的版本可以保证在旧版本设备中，不使用不支持的SearchView。在我们这个例子中，这一操作在onCreateOptionsMenu())方法中:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        SearchManager searchManager =
                (SearchManager) getSystemService(Context.SEARCH_SERVICE);
        SearchView searchView =
                (SearchView) menu.findItem(R.id.search).getActionView();
        searchView.setSearchableInfo(
                searchManager.getSearchableInfo(getComponentName()));
        searchView.setIconifiedByDefault(false);
    }
    return true;
}
```

使得你的App内容可被Google搜索

> 编写:Lin-H - 原文:http://developer.android.com/training/app-indexing/index.html

随着移动app变得越来越普遍，用户不仅仅从网站上查找相关信息，也在他们安装的app上查找。你可以使Google能够抓取你的app内容，当内容与你自己的网页一致时，Google搜索的结果会将你的app作为结果展示给用户。

通过为你的activity提供intent filter，可以使Google搜索展示你的app中特定的内容。Google搜索应用索引(Google Search app indexing)通过在用户搜索结果的网页链接旁附上相关的app内容链接，补充了这一功能。使用移动设备的用户可以在他们的搜索结果中点击链接来打开你的app，使他们能够直接浏览你的app中的内容，而不需要打开网页。

要启用Google搜索应用索引，你需要把有关app与网页之间联系的信息提供给Google。这个过程包括下面几个步骤:

1. 通过在你的app manifest中添加intent filter来开启链接到你的app中指定内容的深度链接。
2. 在你的网站中的相关页面或Sitemap文件中为这些链接添加注解。
3. 选择允许谷歌爬虫(Googlebot)在Google Play store中通过APK抓取，建立app内容索引。在早期采用者计划(early adopter program)中作为参与者加入时，会自动选择允许。

这节课程，会向你展示如何启用深度链接和建立应用内容索引，使用户可以从移动设备搜索结果直接打开此内容。

Lessons

- 为App内容开启深度链接

  演示如何添加intent filter来启用链接app内容的深度链接

- 为索引指定App内容

  演示如何给网站的metadata添加注解，使Google的算法能为app内容建立索引

为App内容开启深度链接

> 编写:Lin-H - 原文:http://developer.android.com/training/app-indexing/deep-linking.html

为使Google能够抓取你的app内容，并允许用户从搜索结果进入你的app，你必须给你的app manifest中相关的activity添加intent filter。这些intent filter能使深度链接与你的任何activity相连。例如，用户可以在购物app中，点击一条深度链接来浏览一个介绍了自己所搜索的产品的页面。

为你的深度链接添加Intent filter

要创建一条与你的app内容相连的深度链接，添加一个包含了以下这些元素和属性值的intent filter到你的manifest中:

`<action>`

指定ACTION_VIEW的操作，使得Google搜索可以触及intent filter。

`<data>`

添加一个或多个 `<data>` 标签，每一个标签代表一种activity对URI格式的解析， `<data>` 必须至少包含android:scheme属性。

你可以添加额外的属性来改善activity所接受的URI类型。例如，你或许有几个activity可以接受相似的URI，它们仅仅是路径名不同。在这种情况下，使用android:path属性或它的变形(`pathPattern` 或 `pathPrefix`)，使系统能辨别对不同的URI路径应该启动哪个activity。

`<category>`

包括BROWSABLE category。BROWSABLE category对于使intent filter能被浏览器访问是必要的。没有这个category，在浏览器中点击链接无法解析到你的app。DEFAULT category是可选的，但建议添加。没有这个category，activity只能够使用app组件名称以显示(explicit)intent启动。

下面的一段XML代码向你展示，你应该如何在manifest中为深度链接指定一个intent filter。URI “example://gizmos” 和 “http://www.example.com/gizmos” 都能够解析到这个activity。

```
<activity
    android:name="com.example.android.GizmosActivity"
    android:label="@string/title_gizmos" >
    <intent-filter android:label="@string/filter_title_viewgizmos">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <!-- 接受以"example://gizmos”开头的 URIs  -->
        <data android:scheme="example"
              android:host="gizmos" />
        <!-- 接受以"http://www.example.com/gizmos”开头的 URIs  -->
        <data android:scheme="http"
              android:host="www.example.com"
              android:pathPrefix="gizmos" />
    </intent-filter>
</activity>
```

当你把包含有指定activity内容的URI的intent filter添加到你的app manifest后，Android就可以在你的app运行时，为app与匹配URI的Intent建立路径。

> **Note:** 对一个URI pattern，intent filter可以只包含一个单一的 `data` 元素，创建不同的intent filter来匹配额外的URI pattern。

学习更多关于定义intent filter，见Allow Other Apps to Start Your Activity

从传入的intent读取数据

一旦系统通过一个intent filter启动你的activity，你可以使用由Intent提供的数据来决定需要处理什么。调用getData()和getAction()方法来取出传入Intent中的数据与操作。你可以在activity生命周期的任何时候调用这些方法，但一般情况下你应该在前期回调如onCreate()或onStart()中调用。

这个是一段代码，展示如何从Intent中取出数据:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Intent intent = getIntent();
    String action = intent.getAction();
    Uri data = intent.getData();
}
```

遵守下面这些惯例来提高用户体验:

- 深度链接应直接为用户打开内容，不需要任何提示，插播式广告页和登录页面。要确保用户能看到app的内容，即使之前从没打开过这个应用。当用户从启动器打开app时，可以在操作结束后给出提示。这个准则也同样适用于网站的first click free体验。
- 遵循Navigation with Back and Up中的设计指导，来使你的app能够满足用户通过深度链接进入app后，向后导航的需求。

测试你的深度链接

你可以使用Android Debug Bridge和activity管理(am)工具来测试你指定的intent filter URI，能否正确解析到正确的app activity。你可以在设备或者模拟器上运行adb命令。

测试intent filter URI的一般adb语法是:

```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d <URI> <PACKAGE>
```

例如，下面的命令试图浏览与指定URI相关的目标app activity。

```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d "example://gizmos" com.example.android
```

为索引指定App内容

> 编写:Lin-H - 原文: http://developer.android.com/training/app-indexing/enabling-app-indexing.html

Google的网页爬虫机器(Googlebot)会抓取页面，并为Google搜索引擎建立索引，也能为你的Android app内容建立索引。通过选择加入这一功能，你可以允许Googlebot通过抓取在Google Play Store中的APK内容，为你的app内容建立索引。要指出哪些app内容你想被Google索引，只需要添加链接元素到现有的Sitemap文件，或添加到你的网站中每个页面的 `<head>` 元素中，以相同的方式为你的页面添加。

你所共享给Google搜索的深度链接必须按照下面的URI格式:

```
android-app://<package_name>/<scheme>/<host_path>
```

构成URI的各部分是:

- **package_name** 代表在Google Play Developer Console中所列出来的你的APK的包名。
- **scheme** 匹配你的intent filter的URI方案。
- **host_path** 找出你的应用中所指定的内容。

下面的几节叙述如何添加一个深度链接URI到你的Sitemap或网页中。

添加深度链接(Deep link)到你的Sitemap

要在你的Sitemap中为Google搜索app索引(Google Search app indexing)添加深度链接的注解，使用 `<xhtml:link>` 标签，并指定用作替代URI的深度链接。

例如，下面一段XML代码向你展示如何使用 `<loc>` 标签指定一个链接到你的页面的链接，以及如何使用 `<xhtml:link>` 标签指定链接到你的Android app的深度链接。

```
<?xml version="1.0" encoding="UTF-8" ?>
<urlset
    xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
    xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <url>
        <loc>example://gizmos</loc>
            <xhtml:link
                rel="alternate"
                href="android-app://com.example.android/example/gizmos" />
    </url>
    ...
</urlset>
```

添加深度链接到你的网页中

除了在你的Sitemap文件中，为Google搜索app索引指定深度链接外，你还可以在你的HTML标记网页中给深度链接添加注解。你可以在 `<head>` 标签内这么做，为每一个页面添加一个 `<link>` 标签，并指定用作替代URI的深度链接。

例如，下面的一段HTML代码向你展示如何在页面中指定一个URL为 `example://gizmos` 的相应的深度链接。

```
<html>
<head>
    <link rel="alternate"
          href="android-app://com.example.android/example/gizmos" />
    ...
</head>
<body> ... </body>
```

允许Google通过你的app抓取URL请求

一般来说，你可以通过使用robots.txt文件，来控制Googlebot如何抓取你网站上的公开访问的URL。当Googlebot为你的app内容建立索引后，你的app可以把HTTP请求当做一般操作。但是，这些请求会被视为从Googlebot发出，发送到你的服务器上。因此，你必须正确配置你的服务器上的 `robots.txt` 文件来允许这些请求。

例如，下面的 `robots.txt` 指示向你展示，如何允许你网站上的特定目录(如 `/api/`)能被你的app访问，并限制Googlebot访问你的网站上的其他目录。

```
User-Agent: Googlebot
Allow: /api/
Disallow: /
```

学习更多关于如何修改`robots.txt`，来控制页面抓取，详见Controlling Crawling and Indexing Getting Started。

Android界面设计

These classes teach you how to build a user interface using Android layouts for all types of devices. Android provides a flexible framework for UI design that allows your app to display different layouts for different devices, create custom UI widgets, and even control aspects of the system UI outside your app's window.

Designing for Multiple Screens

How to build a user interface that's flexible enough to fit perfectly on any screen and how to create different interaction patterns that are optimized for different screen sizes.

Creating Custom Views

How to build custom UI widgets that are interactive and smooth.

Creating Backward-Compatible UIs

How to use UI components and other APIs from the more recent versions of Android while remaining compatible with older versions of the platform.

Implementing Accessibility

How to make your app accessible to users with vision impairment or other physical disabilities.

Managing the System UI

How to hide and show status and navigation bars across different versions of Android, while managing the display of other screen components.

Creating Apps with Material Design

How to implement material design on Android.

为多屏幕设计

> 编写:riverfeng - 原文:http://developer.android.com/training/multiscreen/index.html

从小屏手机到大屏电视，android拥有数百种不同屏幕尺寸的设备。因此，设计兼容不同屏幕尺寸的应用程序满足不同的用户体验就变得非常重要。

但是，只是单纯的兼容不同的设备类型是远远不够的。每个不同的屏幕尺寸都给用户体验带来不同的可能性和挑战。所以，为了充分的满足和打动用户，你的应用不仅要支持多屏幕，更要针对每个屏幕配置优化你的用户体验。

这个课程就将教你如何针对不同屏幕配置来优化你的UI。

本课程提供了一个简单的示例NewsReader。这个示例中每节课的代码展示了如何更好的优化多屏幕适配，你也可以将这个示例中的代码运用到你自己的项目中。

> Note：这节课中相关的例子为了兼容android 3.0以下的版本使用了support library中的Fragment相关APIs。在使用该示例前，请先确定support library已经添加到你的应用中。

Lessons

- 支持不同屏幕尺寸

  这节课程将引导你如何设计适配多种不同尺寸的布局（通过使用灵活的尺寸规格guige（dimensions），相对布局（RelativeLayout），屏幕尺寸和方向限定（qualifiers），别名过滤器（alias filter）和点9图片）。

- 支持不同的屏幕密度

  这节课程将演示如何支持不同像素密度的屏幕（使用密度独立像素（dip）以及为不同的密度提供合适的位图（bitmap））。

- 实现自适应UI流（Flows）

  这节课将演示如何以UI流（flow）的方式来适配一些屏幕大小/密度组合（动态布局运行时检测，响应当前布局，处理屏幕配置变化）。

支持不同的屏幕大小

> 编写:riverfeng - 原文:http://developer.android.com/training/multiscreen/screensizes.html

这节课教你如何通过以下几种方式支持多屏幕：

1、确保你的布局能自适应屏幕

2、根据你的屏幕配置提供合适的UI布局

3、确保正确的布局适合正确的屏幕。

4、提供缩放正确的位图（bitmap）

使用“wrap_content”和“match_parent”

为了确保你的布局能灵活的适应不同的屏幕尺寸，针对一些view组件，你应该使用wrap_content和match_parent来设置他们的宽和高。如果你使用了wrap_content，view的宽和高会被设置为该view所包含的内容的大小值。如果是match_parent（在API 8之前是fill_parent）则会匹配该组件的父控件的大小。

通过使用wrap_content和match_parent尺寸值代替硬编码的尺寸，你的视图将分别只使用控件所需要的空间或者被拓展以填充所有有效的空间。比如：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
                  android:id="@+id/linearLayout1"
                  android:gravity="center"
                  android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
                   android:layout_height="wrap_content"
                   android:layout_width="wrap_content"
                   android:src="@drawable/logo"
                   android:paddingRight="30dp"
                   android:layout_gravity="left"
                   android:layout_weight="0" />
        <View android:layout_height="wrap_content"
              android:id="@+id/view1"
              android:layout_width="wrap_content"
              android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
                android:background="@drawable/button_bg"
                android:layout_height="match_parent"
                android:layout_weight="0"
                android:layout_width="120dp"
                style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

注意上面的例子使用wrap_content和match_parent来指定组件尺寸而不是使用固定的尺寸。这样就能使你的布局正确的适配不同的屏幕尺寸和屏幕方向（这里的配置主要是指屏幕的横竖屏切换）。

例如，下图演示的就是该布局在竖屏和横屏模式下的效果，注意组件的尺寸是自动适应宽和高的。

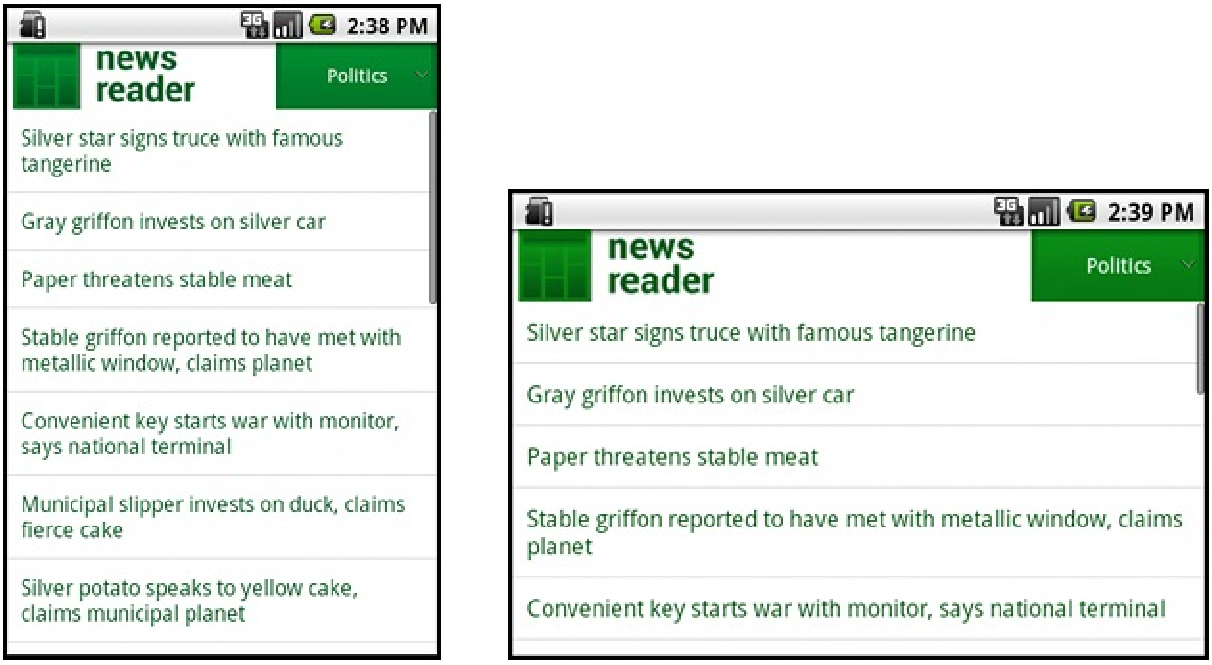


图1：News Reader示例app（左边竖屏，右边横屏）。

使用相对布局（RelativeLayout）

你可以使用LinearLayout以及wrap_content和match_parent组合来构建复杂的布局，但是LinearLayout却不允许你精准的控制它子view的关系，子view在LinearLayout中只能简单一个接一个的排成行。如果你需要你的子view不只是简简单单的排成行的排列，更好的方法是使用RelativeLayout，它允许你指定你布局中控件与控件之间的关系，比如，你可以指定一个子view在左边，另一个则在屏幕的右边。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Type here:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dp"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
        android:layout_alignTop="@id/ok"
        android:text="Cancel" />
</RelativeLayout>
```



图2：QVGA（小尺寸屏幕）屏幕下截图



图3：WSVGA（大尺寸屏幕）屏幕下截图

> 注意：尽管组件的尺寸发生了变化，但是它的子view之间的空间关系还是通过RelativeLayout.LayoutParams已经指定好了。

使用尺寸限定词

（译者注：这里的限定词主要是指在编写布局文件时，将布局文件放在加上类似large，sw600dp等这样限定词的文件夹中，以此来告诉系统根据屏幕选择对应的布局文件，比如下面例子的layout-large文件夹）

从上一节的学习里程中，我们知道如何编写灵活的布局或者相对布局，它们都能通过拉伸或者填充控件来适应不同的屏幕，但是它们却不能为每个不同屏幕尺寸提供最好的用户体验。因此，你的应用不应该只是实现灵活的布局，同时也应该为不同的屏幕配置提供几种不同的布局方式。你可以通过配置限定（configuration qualifiers）来做这件事情，它能在运行时根据你当前设备的配置（比如不同的屏幕尺寸设计了不同的布局）来选择合适的布局资源。

比如，很多应用都为大屏幕实现了“两个窗格”模式（应用可能在一个窗格中实现一个list的item，另外一个则实现list的content），平板和电视都是大到能在一个屏幕上适应两个窗格，但是手机屏幕却只能分别显示。所以，如果你想实现这些布局，你就需要以下文件：

res/layout/main.xml.单个窗格（默认）布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout-large/main.xml,两个窗格布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

注意第二个布局文件的目录名字“large qualifier”，在大尺寸的设备屏幕时（比如7寸平板或者其他大屏幕的设备）就会选择该布局文件，而其他比较小的设备则会选择没有限定词的另一个布局（也就是第一个布局文件）。

使用最小宽度限定词

在Android 3.2之前，开发者还有一个困难，那就是Android设备的“large”屏幕尺寸，其中包括Dell Streak（设备名称），老版Galaxy Tab和一般的7寸平板，有很多的应用都想针对这些不同的设备（比如5和7寸的设备）定义不同的布局，但是这些设备都被定义为了large尺寸屏幕。也是因为这个，所以Android在3.2的时候开始使用最小宽度限定词。

最小宽度限定词允许你根据设备的最小宽度（dp单位）来指定不同布局。比如，传统的7寸平板最小宽度为600dp，如果你希望你的UI能够在这样的屏幕上显示两个窗格（不是一个窗格显示在小屏幕上），你可以使用上节中提到的使用同样的两个布局文件。不同的是，使用sw600来指定两个方框的布局使用在最小宽度为600dp的设备上。

res/layout/main.xml,单个窗格（默认）布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout-sw600dp/main.xml,两个方框布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

这样意味着当你的设备的最小宽度等于600dp或者更大时，系统选择layout-sw600dp/main.xml（两个窗格）的布局，而小一点的屏幕则会选择layout/main.xml（单个窗格）的布局。 然而，在3.2之前的设备上，这样做并不是很好的选择。因为3.2之前还没有将sw600dp作为一个限定词出现，所以，你还是需要使用large限定词来做。因此，你还是应该要有一个布局文件名为res/layout-large/main.xml，和res/layout-sw600dp/main.xml一样。在下一节中，你将学到如何避免像这样出现重复的布局文件。

使用布局别名

最小宽度限定词只能在android3.2或者更高的版本上使用。因此，你还是需要使用抽象尺寸（small，normal，large，xlarge）来兼容以前的版本。比如，你想要将你的UI设计为在手机上只显示一个方框的布局，而在7寸平板或电视，或者其他大屏幕设备上显示多个方框的布局，你可能得提供这些文件：

- res/layout/main.xml：单个窗格布局
- res/layout-large：多个窗格布局

- res/layout-sw600dp：多个窗格布局

最后两个文件都是一样的，因为其中一个将会适配Android3.2的设备，而另外一个则会适配其他Android低版本的平板或者电视。 为了避免这些重复的文件（维护让人感觉头痛就是因为这个），你可以使用别名文件。比如，你可以定义如下布局：

- res/layout/main.xml，单个方框布局
- res/layout/main_twopans.xml，两个方框布局

然后添加这两个文件：

- res/values-large/layout.xml：

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

- res/values-sw600dp/layout.xml：

  ```
  <resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
  </resources>
  ```

  最后两个文件拥有相同的内容，但它们并没有真正意义上的定义布局。它们只是将main_twopanes设置成为了别名main，它们分别处在large和sw600dp选择器中，所以它们能适配Android任何版本的平板和电视（在3.2之前平板和电视可以直接匹配large，而3.2或者以上的则匹配sw600dp）。

使用方向限定词

有一些布局不管是在横向还是纵向的屏幕配置中都能显示的非常好，但是更多的时候，适当的调整一下会更好。在News Reader应用例子中，以下是布局在不同屏幕尺寸和方向的行为：

- 小屏幕，纵向：一个窗格加logo
- 小屏幕，横向：一个窗格加logo
- 7寸平板，纵向：一个窗格加action bar
- 7寸平板，横向：两个宽窗格加action bar
- 10寸平板，纵向：两个窄窗格加action bar
- 10寸平板，横向：两个宽窗格加action bar
- 电视，横向：两个宽窗格加action bar

这些每个布局都会在res/layout目录下定义一个xml文件，如此，应用就能根据屏幕配置的变化根据别名匹配到对应的布局来适应屏幕。

res/layout/onepane.xml：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/onepane_with_bar.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
                  android:id="@+id/linearLayout1"
                  android:gravity="center"
                  android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
                   android:layout_height="wrap_content"
                   android:layout_width="wrap_content"
                   android:src="@drawable/logo"
                   android:paddingRight="30dp"
                   android:layout_gravity="left"
                   android:layout_weight="0" />
        <View android:layout_height="wrap_content"
              android:id="@+id/view1"
              android:layout_width="wrap_content"
              android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
                android:background="@drawable/button_bg"
                android:layout_height="match_parent"
                android:layout_weight="0"
                android:layout_width="120dp"
                style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/twopanes.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

res/layout/twopanes_narrow.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="200dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

现在所有可能的布局我们都已经定义了，唯一剩下的问题是使用方向限定词来匹配对应的布局给屏幕。这时候，你就可以使用布局别名的功能了：

res/values/layouts.xml：

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-sw600dp-land/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-sw600dp-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-large-land/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

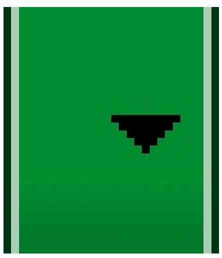
res/values-large-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes_narrow</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

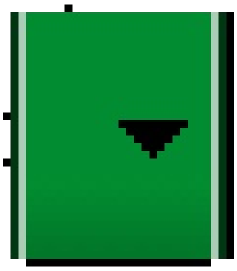
使用.9.png图片

支持不同的屏幕尺寸同时也意味着你的图片资源也必须能兼容不同的屏幕尺寸。比如，一个button的背景图片就必须要适应该button的各种形状。

如果你在使用组件时可以改变图片的大小，你很快就会发现这是一个不明确的选择。因为运行的时候，图片会被拉伸或者压缩（这样容易造成图片失真）。避免这种情况的解决方案就是使用点9图片，这是一种能够指定哪些区域能够或者不能够拉伸的特殊png文件。

因此，在设计的图片需要与组件一起变大变小时，一定要使用点9.若要将位图转换为点9，你可以用一个普通的图片开始（下图，是在4倍变焦情况下的图片显示）。

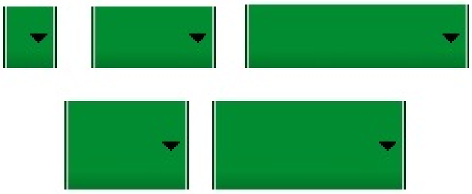
你可以通过sdk中的draw9patch程序（位于tools/directory目录下）来画点9图片。通过沿左侧和顶部边框绘制像素来标记应该被拉伸的区域。也可以通过沿右侧和底部边界绘制像素来标记。就像下图所示一样：



请注意，上图沿边界的黑色像素。在顶部边框和左边框的那些表明图像的可拉伸区域，右边和底部边框则表示内容应该放置的地方。

此外，注意.9.png这个格式，你也必须用这个格式，因为系统会检测这是一个点9图片而不是一个普通PNG图片。

当你将这个应用到组件的背景的时候（通过设置android:background="@drawable/button"），android框架会自动正确的拉伸图像以适应按钮的大小，下图就是各种尺寸中的显示效果：



兼容不同的屏幕密度

> 编写:riverfeng - 原文:http://developer.android.com/training/multiscreen/screendensities.html

这节课将教你如何通过提供不同的资源和使用独立分辨率（dp）来支持不同的屏幕密度。

使用密度独立像素（dp）

设计布局时，要避免使用绝对像素（absolutepixels）定义距离和尺寸。使用像素单位来定义布局大小是有问题的。因为，不同的屏幕有不同的像素密度，所以，同样单位的像素在不同的设备上会有不同的物理尺寸。因此，在指定单位的时候，通常使用dp或者sp。一个dp代表一个密度独立像素，也就相当于在160 dpi的一个像素的物理尺寸，sp也是一个基本的单位，不过它主要是用在文本尺寸上（它也是一种尺寸规格独立的像素），所以，你在定义文本尺寸的时候应该使用这种规格单位（不要使用在布尺寸上）。

例如，当你是定义两个view之间的空间时，应该使用dp而不是px：

```
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/clickme"
    android:layout_marginTop="20dp" />
```

当指定文本尺寸时，始终应该使用sp：

```
<TextView android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="20sp" />
```

提供可供选择的图片

因为Android能运行在很多不同屏幕密度的设备上，所以，你应该针对不同的设备密度提供不同的bitmap资源：小屏幕（low），medium（中），high（高）以及超高（extra-high）密度。这将能帮助你在所有的屏幕密度中得到非常好的图形质量和性能。

为了提供更好的用户体验，你应该使用以下几种规格来缩放图片大小，为不同的屏幕密度提供相应的位图资源：

```
xhdpi:2.0
hdpi:1.5
mdpi:1.0(标准线)
ldpi:0.75
```

这也就意味着如果在xhdpi设备上你需要一个200x200的图片，那么你则需要一张150x150的图片用于hdpi，100x100的用于mdpi以及75x75的用户ldpi设备。

然后将这些图片资源放到res/对应的目录下面，系统会自动根据当前设备屏幕密度自动去选择合适的资源进行加载：

```
MyProject/
  res/
    drawable-xhdpi/
        awesomeimage.png
    drawable-hdpi/
        awesomeimage.png
    drawable-mdpi/
        awesomeimage.png
    drawable-ldpi/
        awesomeimage.png
```

这样放置图片资源后，不论你什么时候使用@drawable/awesomeimage，系统都会给予屏幕的dp来选择合适的图片。

如果你想知道更多关于如何为你的应用程序创建icon资源，你可以看看Icon设计指南Icon Design Guidelines.

实现自适应UI流（Flows）

> 编写:riverfeng - 原文:http://developer.android.com/training/multiscreen/adaptui.html

根据当前你的应用显示的布局，它的UI流可能会不一样。比如，当你的应用是双窗格模式，点击左边窗格的条目（item）时，内容（content）显示在右边窗格中。如果是单窗格模式中，当你点击某个item的时候，内容则显示在一个新的activity中。

确定当前布局

由于每种布局的实现会略有差别，首先你可能要确定用户当前可见的布局是哪一个。比如，你可能想知道当前用户到底是处于“单窗格”的模式还是“双窗格”的模式。你可以通过检查指定的视图（view）是否存在和可见来实现：

```
public class NewsReaderActivity extends FragmentActivity {
    boolean mIsDualPane;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);

        View articleView = findViewById(R.id.article);
        mIsDualPane = articleView != null &&
                        articleView.getVisibility() == View.VISIBLE;
    }
}
```

> 注意：使用代码查询id为“article”的view是否可见比直接硬编码查询指定的布局更加的灵活。

另一个关于如何适配不同组件是否存在的例子，是在组件执行操作之前先检查它是否是可用的。比如，在News Reader示例中，有一个按钮点击后打开一个菜单，但是这个按钮仅仅只在Android3.0之后的版本中才能显示（因为这个功能被ActionBar代替，在API 11+中定义）。所以，在给这个按钮添加事件之间，你可以这样做：

```
Button catButton = (Button) findViewById(R.id.categorybutton);
OnClickListener listener = /* create your listener here */;
if (catButton != null) {
    catButton.setOnClickListener(listener);
}
```

根据当前布局响应

一些操作会根据当前的布局产生不同的效果。比如，在News Reader示例中，当你点击标题（headlines）列表中的某一条headline时，如果你的UI是双窗格模式，内容会显示在右边的窗格中，如果你的UI是单窗格模式，会启动一个分开的Activity并显示：

```
@Override
public void onHeadlineSelected(int index) {
    mArtIndex = index;
    if (mIsDualPane) {
        /* display article on the right pane */
        mArticleFragment.displayArticle(mCurrentCat.getArticle(index));
    } else {
        /* start a separate activity */
        Intent intent = new Intent(this, ArticleActivity.class);
        intent.putExtra("catIndex", mCatIndex);
        intent.putExtra("artIndex", index);
        startActivity(intent);
    }
}
```

同样，如果你的应用处于多窗格模式，那么它应该在导航栏中设置带有选项卡的action bar。而如果是单窗格模式，那么导航栏应该设置为spinner widget。所以，你的代码应该检查哪个方案是最合适的：

```
final String CATEGORIES[] = { "Top Stories", "Politics", "Economy", "Technology" };

public void onCreate(Bundle savedInstanceState) {
    ....
    if (mIsDualPane) {
        /* use tabs for navigation */
        actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS);
        int i;
        for (i = 0; i < CATEGORIES.length; i++) {
            actionBar.addTab(actionBar.newTab().setText(
                CATEGORIES[i]).setTabListener(handler));
        }
        actionBar.setSelectedNavigationItem(selTab);
    }
    else {
        /* use list navigation (spinner) */
        actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_LIST);
        SpinnerAdapter adap = new ArrayAdapter(this,
                R.layout.headline_item, CATEGORIES);
        actionBar.setListNavigationCallbacks(adap, handler);
    }
}
```

在其他Activity中复用Fragment

在多屏幕设计时经常出现的情况是：在一些屏幕配置上设计一个窗格，而在其他屏幕配置上启动一个独立的Activity。例如，在News Reader中，新闻内容文字在大屏幕上市显示在屏幕右边的方框中，而在小屏幕中，则是由单独的activity显示的。

像这样的情况，你就应该在不同的activity中使用同一个Fragment，以此来避免代码的重复，而达到代码复用的效果。比如，ArticleFragment在双窗格模式下是这样用的：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

在小屏幕中，它又是如下方式被复用的（没有布局文件）：

```
ArticleFragment frag = new ArticleFragment();
getSupportFragmentManager().beginTransaction().add(android.R.id.content, frag).commit(
);
```

当然，如果将这个fragment定义在XML布局文件中，也有同样的效果，但是在这个例子中，则没有必要，因为这个article fragment是这个activity的唯一组件。

当你在设计fragment的时候，非常重要的一点：不要为某个特定的activity设计耦合度高的fragment。通常的做法是，通过定义抽象接口，并在接口中定义需要与该fragment进行交互的activity的抽象方法，然后与该fragment进行交互的activity实现这些抽象接口方法。

例如，在News Reader中，HeadlinesFragment就很好的诠释了这一点：

```
public class HeadlinesFragment extends ListFragment {
    ...
    OnHeadlineSelectedListener mHeadlineSelectedListener = null;

    /* Must be implemented by host activity */
    public interface OnHeadlineSelectedListener {
        public void onHeadlineSelected(int index);
    }
    ...

    public void setOnHeadlineSelectedListener(OnHeadlineSelectedListener listener) {
        mHeadlineSelectedListener = listener;
    }
}
```

然后，当用户选择了一个headline item之后，fragment将通知对应的activity指定监听事件（而不是通过硬编码的方式去通知）：

```
public class HeadlinesFragment extends ListFragment {
    ...
    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position, long id) {
        if (null != mHeadlineSelectedListener) {
            mHeadlineSelectedListener.onHeadlineSelected(position);
        }
    }
    ...
}
```

这种技术在支持平板与手持设备(Supporting Tablets and Handsets)有更加详细的介绍。

处理屏幕配置变化

如果使用的是单独的activity来实现你界面的不同部分，你需要注意的是，屏幕变化（如旋转变化）的时候，你也应该根据屏幕配置的变化来保持你的UI布局的一致性。

例如，在传统的Android3.0或以上版本的7寸平板上，News Reader示例在竖屏的时候使用独立的activity显示文章内容，而在横屏的时候，则使用两个窗格模式（即内容显示在右边的方框中）。 这也就意味着，当用户在竖屏模式下观看文章的时候，你需要检测屏幕是否变成了横屏，如果改变了，则结束当前activity并返回到主activity中，这样，content就能显示在双窗格模式布局中。

```
public class ArticleActivity extends FragmentActivity {
    int mCatIndex, mArtIndex;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCatIndex = getIntent().getExtras().getInt("catIndex", 0);
        mArtIndex = getIntent().getExtras().getInt("artIndex", 0);

        // If should be in two-pane mode, finish to return to main activity
        if (getResources().getBoolean(R.bool.has_two_panes)) {
            finish();
            return;
        }
        ...
}
```

创建自定义View

> 编写:kesenhoo - 原文:http://developer.android.com/training/custom-views/index.html

Android的framework有大量的Views用来与用户进行交互并显示不同种类的数据。但是有时候你的程序有个特殊的需求，而Android内置的views组件并不能实现。这一章节会演示如何创建你自己的views，并使得它们是robust与reusable的。

依赖和要求

Android 2.1 (API level 7) 或更高

你也可以看

- Custom Components
- Input Events
- Property Animation
- Hardware Acceleration
- Accessibility developer guide

Sample

CustomView.zip

Lesson

- **创建一个View类**

  创建一个像内置的view，有自定义属性并支持ADT layout编辑器。

- **自定义Drawing**

  使用Android graphics系统使你的view拥有独特的视觉效果。

- **使得View是可交互的**

  用户期望view对操作反应流畅自然。这节课会讨论如何使用gesture detection, physics, 和 animation使你的用户界面有专业的水准。

- **优化View**

  不管你的UI如何的漂亮，如果不能以高帧率流畅运行，用户也不会喜欢。学习如何避免一般的性能问题，和如何使用硬件加速来使你的自定义图像运行更流畅。

创建自定义的View类

> 编写:kesenhoo - 原文:http://developer.android.com/training/custom-views/create-view.html

设计良好的类总是相似的。它使用一个好用的接口来封装一个特定的功能，它有效的使用CPU与内存，等等。为了成为一个设计良好的类，自定义的view应该:

- 遵守Android标准规则。
- 提供自定义的风格属性值并能够被Android XML Layout所识别。
- 发出可访问的事件。
- 能够兼容Android的不同平台。

Android的framework提供了许多基类与XML标签用来帮助你创建一个符合上面要求的View。这节课会介绍如何使用Android framework来创建一个view的核心功能。

继承一个View

Android framework里面定义的view类都继承自View。你自定义的view也可以直接继承View，或者你可以通过继承既有的一个子类(例如Button)来节约一点时间。

为了让Android Developer Tools能够识别你的view，你必须至少提供一个constructor，它包含一个Contenx与一个AttributeSet对象作为参数。这个constructor允许layout editor创建并编辑你的view的实例。

```
class PieChart extends View {
    public PieChart(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

定义自定义属性

为了添加一个内置的View到你的UI上，你需要通过XML属性来指定它的样式与行为。良好的自定义views可以通过XML添加和改变样式，为了让你的自定义的view也有如此的行为，你应该:

- 为你的view在资源标签下定义自设的属性
- 在你的XML layout中指定属性值
- 在运行时获取属性值
- 把获取到的属性值应用在你的view上

这一节讨论如何定义自定义属性以及指定属性值，下一节将会实现在运行时获取属性值并将它应用。

为了定义自设的属性，添加 资源到你的项目中。放置于res/values/attrs.xml文件中。下面是一个attrs.xml文件的示例:

```
<resources>
   <declare-styleable name="PieChart">
       <attr name="showText" format="boolean" />
       <attr name="labelPosition" format="enum">
           <enum name="left" value="0"/>
           <enum name="right" value="1"/>
       </attr>
   </declare-styleable>
</resources>
```

上面的代码声明了2个自设的属性，**showText**与**labelPosition**，它们都归属于PieChart的项目下的styleable实例。styleable实例的名字，通常与自定义的view名字一致。尽管这并没有严格规定要遵守这个convention，但是许多流行的代码编辑器都依靠这个命名规则来提供statement completion。

一旦你定义了自设的属性，你可以在layout XML文件中使用它们，就像内置属性一样。唯一不同的是你自设的属性是归属于不同的命名空间。不是属于 `http://schemas.android.com/apk/res/android` 的命名空间，它们归属于 `http://schemas.android.com/apk/res/[your package name]` 。例如，下面演示了如何为PieChart使用上面定义的属性：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
 <com.example.customviews.charting.PieChart
     custom:showText="true"
     custom:labelPosition="left" />
</LinearLayout>
```

为了避免输入长串的namespace名字，示例上面使用了 `xmlns` 指令，这个指令可以指派 `custom` 作为 `http://schemas.android.com/apk/res/com.example.customviews` namespace的别名。你也可以选择其他的别名作为你的namespace。

请注意，如果你的view是一个inner class，你必须指定这个view的outer class。同样的，如果PieChart有一个inner class叫做PieView。为了使用这个类中自设的属性，你应该使用com.example.customviews.charting.PieChart$PieView.

应用自定义属性

当view从XML layout被创建的时候，在xml标签下的属性值都是从resource下读取出来并传递到view的constructor作为一个AttributeSet参数。尽管可以从AttributeSet中直接读取数值，可是这样做有些弊端：

- 拥有属性的资源并没有经过解析
- Styles并没有运用上

> 翻译注：通过 attrs 的方法是可以直接获取到属性值的，但是不能确定值类型，如:
>
> ```
> String title = attrs.getAttributeValue(null, "title");
> int resId = attrs.getAttributeResourceValue(null, "title", 0);
> title = context.getText(resId));
> ```
>
> 都能获取到 "title" 属性，但你不知道值是字符串还是resId，处理起来就容易出问题，下面的方法则能在编译时就发现问题

取而代之的是，通过obtainStyledAttributes()来获取属性值。这个方法会传递一个TypedArray对象，它是间接referenced并且styled的。

Android资源编译器帮你做了许多工作来使调用obtainStyledAttributes()更简单。对res目录里的每一个 `<declare-styleable>` 资源，自动生成的R.java文件定义了存放属性ID的数组和常量，常量用来索引数组中每个属性。你可以使用这些预先定义的常量来从TypedArray中读取属性。这里就是 `PieChart` 类如何读取它的属性:

```
public PieChart(Context context, AttributeSet attrs) {
   super(context, attrs);
   TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.PieChart,
        0, 0);

   try {
       mShowText = a.getBoolean(R.styleable.PieChart_showText, false);
       mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);
   } finally {
       a.recycle();
   }
}
```

清注意TypedArray对象是一个共享资源，必须被在使用后进行回收。

添加属性和事件

Attributes是一个强大的控制view的行为与外观的方法，但是他们仅仅能够在view被初始化的时候被读取到。为了提供一个动态的行为，需要暴露出一些合适的getter 与setter方法。下面的代码演示了如何使用这个技巧:

```
public boolean isShowText() {
   return mShowText;
}

public void setShowText(boolean showText) {
   mShowText = showText;
   invalidate();
   requestLayout();
}
```

请注意，在setShowText方法里面有调用invalidate()) and requestLayout()). 这两个调用是确保稳定运行的关键。当view的某些内容发生变化的时候，需要调用invalidate来通知系统对这个view进行redraw，当某些元素变化会引起组件大小变化时，需要调用requestLayout方法。调用时若忘了这两个方法，将会导致hard-to-find bugs。

自定义的view也需要能够支持响应事件的监听器。例如，`PieChart` 暴露了一个自定义的事件 `OnCurrentItemChanged` 来通知监听器，用户已经切换了焦点到一个新的组件上。

我们很容易忘记了暴露属性与事件，特别是当你是这个view的唯一用户时。请花费一些时间来仔细定义你的view的交互。一个好的规则是总是暴露任何属性与事件。

设计可访问性

自定义view应该支持广泛的用户群体，包含一些不能看到或使用触屏的残障人士。为了支持残障人士，我们应该：

- 使用 `android:contentDescription` 属性标记输入字段。
- 在适当的时候通过调用 `sendAccessibilityEvent()` 发送访问事件。
- 支持备用控制器，如方向键（D-pad）和轨迹球（trackball）等。

对于创建使用的 views的更多消息, 请参见Android Developers Guide中的 Making Applications Accessible 。

实现自定义View的绘制

> 编写:kesenhoo - 原文:http://developer.android.com/training/custom-view/custom-draw.html

自定义view的最重要的一个部分是自定义它的外观。根据你的程序的需求，自定义绘制可能简单也可能很复杂。这节课会演示一些最常见的操作。

Override onDraw()

重绘一个自定义的view的最重要的步骤是重写onDraw()方法。onDraw()的参数是一个Canvas对象。Canvas类定义了绘制文本，线条，图像与许多其他图形的方法。你可以在onDraw方法里面使用那些方法来创建你的UI。

在你调用任何绘制方法之前，你需要创建一个Paint对象。

创建绘图对象

android.graphics framework把绘制定义为下面两类:

- 绘制什么，由Canvas处理
- 如何绘制，由Paint处理

例如Canvas提供绘制一条直线的方法，Paint提供直线颜色。Canvas提供绘制矩形的方法，Paint定义是否使用颜色填充。简单来说：Canvas定义你在屏幕上画的图形，而Paint定义颜色，样式，字体，

所以在绘制之前，你需要创建一个或者多个Paint对象。在这个PieChart 的例子，是在 `init()` 方法实现的，由constructor调用。

```
private void init() {
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mTextPaint.setColor(mTextColor);
    if (mTextHeight == 0) {
        mTextHeight = mTextPaint.getTextSize();
    } else {
        mTextPaint.setTextSize(mTextHeight);
    }

    mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mPiePaint.setStyle(Paint.Style.FILL);
    mPiePaint.setTextSize(mTextHeight);

    mShadowPaint = new Paint(0);
    mShadowPaint.setColor(0xff101010);
    mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));

    ...
```

刚开始就创建对象是一个重要的优化技巧。Views会被频繁的重新绘制，初始化许多绘制对象需要花费昂贵的代价。在onDraw方法里面创建绘制对象会严重影响到性能并使得你的UI显得卡顿。

处理布局事件

为了正确的绘制你的view，你需要知道view的大小。复杂的自定义view通常需要根据在屏幕上的大小与形状执行多次layout计算。而不是假设这个view在屏幕上的显示大小。即使只有一个程序会使用你的view，仍然是需要处理屏幕大小不同，密度不同，方向不同所带来的影响。

尽管view有许多方法是用来计算大小的，但是大多数是不需要重写的。如果你的view不需要特别的控制它的大小，唯一需要重写的方法是onSizeChanged()).

onSizeChanged()，当你的view第一次被赋予一个大小时，或者你的view大小被更改时会被执行。在onSizeChanged方法里面计算位置，间距等其他与你的view大小值。

当你的view被设置大小时，layout manager(布局管理器)假定这个大小包括所有的view的内边距(padding)。当你计算你的view大小时，你必须处理内边距的值。这段 `PieChart.onSizeChanged()` 中的代码演示该怎么做:

```
        // Account for padding
        float xpad = (float)(getPaddingLeft() + getPaddingRight());
        float ypad = (float)(getPaddingTop() + getPaddingBottom());

        // Account for the label
        if (mShowText) xpad += mTextWidth;

        float ww = (float)w - xpad;
        float hh = (float)h - ypad;

        // Figure out how big we can make the pie.
        float diameter = Math.min(ww, hh);
```

如果你想更加精确的控制你的view的大小，需要重写onMeasure())方法。这个方法的参数是View.MeasureSpec，它会告诉你的view的父控件的大小。那些值被包装成int类型，你可以使用静态方法来获取其中的信息。

这里是一个实现onMeasure()的例子。在这个例子中 `PieChart` 试着使它的区域足够大，使pie可以像它的label一样大:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
   // Try for a width based on our minimum
   int minw = getPaddingLeft() + getPaddingRight() + getSuggestedMinimumWidth();
   int w = resolveSizeAndState(minw, widthMeasureSpec, 1);

   // Whatever the width ends up being, ask for a height that would let the pie
   // get as big as it can
   int minh = MeasureSpec.getSize(w) - (int)mTextWidth + getPaddingBottom() + getPaddi
ngTop();
   int h = resolveSizeAndState(MeasureSpec.getSize(w) - (int)mTextWidth, heightMeasure
Spec, 0);

   setMeasuredDimension(w, h);
}
```

上面的代码有三个重要的事情需要注意:

- 计算的过程有把view的padding考虑进去。这个在后面会提到，这部分是view所控制的。
- 帮助方法resolveSizeAndState()是用来创建最终的宽高值的。这个方法比较 view 的期望值与传递给 onMeasure 方法的 spec 值，然后返回一个合适的View.MeasureSpec值。
- onMeasure()没有返回值。它通过调用setMeasuredDimension()来获取结果。调用这个方法是强制执行的，如果你遗漏了这个方法，会出现运行时异常。

绘图!

每个view的onDraw都是不同的，但是有下面一些常见的操作：

- 绘制文字使用drawText()。指定字体通过调用setTypeface(), 通过setColor()来设置文字颜色.
- 绘制基本图形使用drawRect(), drawOval(), drawArc(). 通过setStyle()来指定形状是否需要filled, outlined.
- 绘制一些复杂的图形，使用Path类. 通过给Path对象添加直线与曲线, 然后使用drawPath()来绘制图形. 和基本图形一样，paths也可以通过setStyle来设置是outlined, filled, both.
- 通过创建LinearGradient对象来定义渐变。调用setShader()来使用LinearGradient。
- 通过使用drawBitmap来绘制图片.

```
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // Draw the shadow
   canvas.drawOval(
           mShadowBounds,
           mShadowPaint
   );

   // Draw the label text
   canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);

   // Draw the pie slices
   for (int i = 0; i < mData.size(); ++i) {
       Item it = mData.get(i);
       mPiePaint.setShader(it.mShader);
       canvas.drawArc(mBounds,
               360 - it.mEndAngle,
               it.mEndAngle - it.mStartAngle,
               true, mPiePaint);
   }

   // Draw the pointer
   canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
   canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

使得View可交互

> 编写:kesenhoo - 原文:http://developer.android.com/training/custom-view/make-interactive.html

绘制UI仅仅是创建自定义View的一部分。你还需要使得你的View能够以模拟现实世界的方式来进行反馈。对象应该总是与现实情景能够保持一致。例如，图片不应该突然消失又从另外一个地方出现，因为在现实世界里面不会发生那样的事情。正确的应该是，图片从一个地方移动到另外一个地方。

用户应该可以感受到UI上的微小变化，并对模仿现实世界的细微之处反应强烈。例如，当用户fling(迅速滑动)一个对象时，应该在开始时感到摩擦带来的阻力，在结束时感到fling带动的动力。应该在滑动开始与结束的时候给用户一定的反馈。

这节课会演示如何使用Android framework的功能来为自定义的View添加那些现实世界中的行为。

处理输入的手势

像许多其他UI框架一样，Android提供一个输入事件模型。用户的动作会转换成触发一些回调函数的事件，你可以重写这些回调方法来定制你的程序应该如何响应用户的输入事件。在Android中最常用的输入事件是touch，它会触发onTouchEvent(android.view.MotionEvent))的回调。重写这个方法来处理touch事件：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  return super.onTouchEvent(event);
}
```

Touch事件本身并不是特别有用。如今的touch UI定义了touch事件之间的相互作用，叫做gestures。例如tapping,pulling,flinging与zooming。为了把那些touch的源事件转换成gestures, Android提供了GestureDetector。

通过传入GestureDetector.OnGestureListener的一个实例构建一个GestureDetector。如果你只是想要处理几种gestures(手势操作)你可以继承GestureDetector.SimpleOnGestureListener，而不用实现GestureDetector.OnGestureListener接口。例如，下面的代码创建一个继承GestureDetector.SimpleOnGestureListener的类，并重写onDown(MotionEvent))。

```
class mListener extends GestureDetector.SimpleOnGestureListener {
   @Override
   public boolean onDown(MotionEvent e) {
       return true;
   }
}
mDetector = new GestureDetector(PieChart.this.getContext(), new mListener());
```

不管你是否使用GestureDetector.SimpleOnGestureListener, 你必须总是实现onDown()方法，并返回true。这一步是必须的，因为所有的gestures都是从onDown()开始的。如果你在onDown()里面返回false，系统会认为你想要忽略后续的gesture,那么GestureDetector.OnGestureListener的其他回调方法就不会被执行到了。一旦你实现了GestureDetector.OnGestureListener并且创建了GestureDetector的实例, 你可以使用你的GestureDetector来中止你在onTouchEvent里面收到的touch事件。

```
@Override
public boolean onTouchEvent(MotionEvent event) {
   boolean result = mDetector.onTouchEvent(event);
   if (!result) {
       if (event.getAction() == MotionEvent.ACTION_UP) {
           stopScrolling();
           result = true;
       }
   }
   return result;
}
```

当你传递一个touch事件到onTouchEvent()时，若这个事件没有被辨认出是何种gesture，它会返回false。你可以执行自定义的gesture-decection代码。

创建基本合理的物理运动

Gestures是控制触摸设备的一种强有力的方式，但是除非你能够产出一个合理的触摸反馈，否则将是违反用户直觉的。一个很好的例子是fling手势，用户迅速的在屏幕上移动手指然后抬手离开屏幕。这个手势应该使得UI迅速的按照fling的方向进行滑动，然后慢慢停下来，就像是用户旋转一个飞轮一样。

但是模拟这个飞轮的感觉并不简单，要想得到正确的飞轮模型，需要大量的物理，数学知识。幸运的是，Android有提供帮助类来模拟这些物理行为。Scroller是控制飞轮式的fling的基类。

要启动一个fling，需调用 `fling()` ，并传入启动速率、x、y的最小值和最大值，对于启动速度值，可以使用GestureDetector计算得出。

```
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocity
Y) {
   mScroller.fling(currentX, currentY, velocityX / SCALE, velocityY / SCALE, minX, min
Y, maxX, maxY);
   postInvalidate();
}
```

> **Note:** 尽管速率是通过GestureDetector来计算的，许多开发者感觉使用这个值使得fling动画太快。通常把x与y设置为4到8倍的关系。

调用fling()时会为fling手势设置物理模型。然后，通过调用定期调用Scroller.computeScrollOffset()来更新Scroller。computeScrollOffset()通过读取当前时间和使用物理模型来计算x和y的位置更新Scroller对象的内部状态。调用getCurrX()和getCurrY()来获取这些值。

大多数view通过Scroller对象的x,y的位置直接到scrollTo()，PieChart例子稍有不同，它使用当前滚动y的位置设置图表的旋转角度。

```
if (!mScroller.isFinished()) {
    mScroller.computeScrollOffset();
    setPieRotation(mScroller.getCurrY());
}
```

Scroller 类会为你计算滚动位置，但是他不会自动把哪些位置运用到你的view上面。你有责任确保View获取并运用到新的坐标。你有两种方法来实现这件事情：

- 在调用fling()之后执行postInvalidate(), 这是为了确保能强制进行重画。这个技术需要每次在onDraw里面计算过scroll offsets(滚动偏移量)之后调用postInvalidate()。
- 使用ValueAnimator在fling是展现动画，并且通过调用addUpdateListener()增加对fling过程的监听。

这个PieChart 的例子使用了第二种方法。这个方法使用起来会稍微复杂一点，但是它更有效率并且避免了不必要的重画的view进行重绘。缺点是ValueAnimator是从API Level 11才有的。因此他不能运用到3.0的系统之前的版本上。

> **Note:** ValueAnimator虽然是API 11才有的，但是你还是可以在最低版本低于3.0的系统上使用它，做法是在运行时判断当前的API Level，如果低于11则跳过。

```
mScroller = new Scroller(getContext(), null, true);
mScrollAnimator = ValueAnimator.ofFloat(0,1);
mScrollAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator valueAnimator) {
        if (!mScroller.isFinished()) {
            mScroller.computeScrollOffset();
            setPieRotation(mScroller.getCurrY());
        } else {
            mScrollAnimator.cancel();
            onScrollFinished();
        }
    }
});
```

使过渡平滑

用户期待一个UI之间的切换是能够平滑过渡的。UI元素需要做到渐入淡出来取代突然出现与消失。Android从3.0开始有提供property animation framework,用来使得平滑过渡变得更加容易。

使用这套动画系统时，任何时候属性的改变都会影响到你的视图，所以不要直接改变属性的值。而是使用ValueAnimator来实现改变。在下面的例子中，在PieChart 中更改选择的部分将导致整个图表的旋转，以至选择的进入选择区内。ValueAnimator在数百毫秒内改变旋转量，而不是突然地设置新的旋转值。

```
mAutoCenterAnimator = ObjectAnimator.ofInt(PieChart.this, "PieRotation", 0);
mAutoCenterAnimator.setIntValues(targetAngle);
mAutoCenterAnimator.setDuration(AUTOCENTER_ANIM_DURATION);
mAutoCenterAnimator.start();
```

如果你想改变的是view的某些基础属性，你可以使用ViewPropertyAnimator ,它能够同时执行多个属性的动画。

```
animate().rotation(targetAngle).setDuration(ANIM_DURATION).start();
```

优化自定义View

> 编写:[kesenhoo](https://github.com/kesenhoo) - 原文:http://developer.android.com/training/custom-views/optimizing-view.html

前面的课程学习到了如何创建设计良好的View，并且能够使之在手势与状态切换时得到正确的反馈。下面要介绍的是如何使得view能够执行更快。为了避免UI显得卡顿，你必须确保动画能够保持在60fps。

Do Less, Less Frequently

为了加速你的view，对于频繁调用的方法，需要尽量减少不必要的代码。先从onDraw开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致GC，从而导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。

你还需要尽可能的减少onDraw被调用的次数，大多数时候导致onDraw都是因为调用了invalidate().因此请尽量减少调用invaildate()的次数。如果可能的话，尽量调用含有4个参数的invalidate()方法而不是没有参数的invalidate()。没有参数的invalidate会强制重绘整个view。

另外一个非常耗时的操作是请求layout。任何时候执行requestLayout()，会使得Android UI系统去遍历整个View的层级来计算出每一个view的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持View的层级是扁平化的，这样对提高效率很有帮助。

如果你有一个复杂的UI，你应该考虑写一个自定义的ViewGroup来执行他的layout操作。与内置的view不同，自定义的view可以使得程序仅仅测量这一部分，这避免了遍历整个view的层级结构来计算大小。这个PieChart 例子展示了如何继承ViewGroup作为自定义view的一部分。PieChart 有子views，但是它从来不测量它们。而是根据他自身的layout法则，直接设置它们的大小。

使用硬件加速

从Android 3.0开始，Android的2D图像系统可以通过GPU (Graphics Processing Unit))来加速。GPU硬件加速可以提高许多程序的性能。但是这并不是说它适合所有的程序。Android framework让你能过随意控制你的程序的各个部分是否启用硬件加速。

参考 Android Developers Guide 中的Hardware Acceleration 来学习如何在application, activity, 或 window 层启用加速。注意除了 Android Guide 的指导之外，你必须要设置你的应用的target API为11，或更高，通过在你的AndroidManifest.xml 文件中增加 < uses-sdk android:targetSdkVersion="11"/> 。

一旦你开启了硬件加速，性能的提示并不一定可以明显察觉到。移动设备的GPU在某些例如scaling,rotating与translating的操作中表现良好。但是对其他一些任务，比如画直线或曲线，则表现不佳。为了充分发挥GPU加速，你应该最大化GPU擅长的操作的数量，最小化GPU不擅长操作的数量。

在下面的例子中，绘制pie是相对来说比较费时的。解决方案是把pie放到一个子view中，并设置View使用LAYER_TYPE_HARDWARE来进行加速。

```
private class PieView extends View {

       public PieView(Context context) {
           super(context);
           if (!isInEditMode()) {
               setLayerType(View.LAYER_TYPE_HARDWARE, null);
           }
       }

       @Override
       protected void onDraw(Canvas canvas) {
           super.onDraw(canvas);

           for (Item it : mData) {
               mPiePaint.setShader(it.mShader);
               canvas.drawArc(mBounds,
                       360 - it.mEndAngle,
                       it.mEndAngle - it.mStartAngle,
                       true, mPiePaint);
           }
       }

       @Override
       protected void onSizeChanged(int w, int h, int oldw, int oldh) {
           mBounds = new RectF(0, 0, w, h);
       }

       RectF mBounds;
   }
```

通过这样的修改以后，PieChart.PieView.onDraw()只会在第一次现实的时候被调用。之后，pie chart会被缓存为一张图片，并通过GPU来进行重画不同的角度。GPU特别擅长这类的事情，并且表现效果突出。

缓存图片到hardware layer会消耗video memory，而video memory又是有限的。基于这样的考虑，仅仅在用户触发scrolling的时候使用 `LAYER_TYPE_HARDWARE` ，在其他时候，使用 `LAYER_TYPE_NONE` 。

创建向后兼容的UI

> 编写:spencer198711 - 原文:http://developer.android.com/training/backward-compatible-ui/index.html

这一课展示了如何以向后兼容的方式使用在新版本的Android上可用的UI组件和API，确保你的应用在之前的版本上依然能够运行。

贯穿整个课程，在Android 3.0被新引入的Action Bar Tabs功能在本课程中作为指导例子，但是你可以在其他UI组件和API功能上运用这种方式。

Sample

http://developer.android.com/shareables/training/TabCompat.zip

Lessons

- **抽象出新的APIs**

  决定你的应用需要的功能和接口。学习如何为你的应用定义面向特定应用的、作为中间媒介并抽象出UI组件具体实现的java接口。

- **代理至新的APIs**

  学习如何创建使用新的APIs的接口的具体实现

- **使用旧的APIs实现新API的效果**

  学习如何创建使用老的APIs的自定义的接口实现

- **使用能感知版本的组件**

  学习如何在运行的时候去选择一个具体的实现，并且开始在你的应用中使用接口。

抽象出新的APIs

> 编写:spencer198711 - 原文:http://developer.android.com/training/backward-compatible-ui/abstracting.html

假如你想使用Action Bar Tabs作为你的应用的顶层导航的主要形式。不幸的是，ActionBar APIs只在Android 3.0（API等级11）之后才能使用。因此，如果你想要在运行之前版本的Android平台的设备上分发你的应用，你需要提供一个支持新的API的实现，同时提供一个回退机制，使得能够使用旧的APIs。

在本课程中，使用了具有面向特定版本实现的抽象类去构建一个tab页形式的用户界面，并以此提供向后兼容性。这一课描述了如何为新的tab API创建一个抽象层，并以此作为构建tab组件的第一步。

为抽象做准备

在Java编程语言中，抽象包含了创建一个或者多个接口或抽象类去隐藏具体的实现细节。在新版本的Android API的情况中，你可以使用抽象去构建能感知版本的组件，这个组件会在新版本的设备上使用当前的APIs，当回退到老的设备上同时存在兼容的APIs。

当使用这种方法时，你首先需要决定哪些要使用的类需要提供向后兼容，然后去根据新类中的public接口去创建抽象类。在创建抽象接口的过程中，你应该尽可能多的为新APIs创建镜像。这会最大化前向兼容性，使得在将来当这些接口不再需要的时候，废弃这些接口会更加容易。

在为新的APIs创建抽象类之后，任何数量的实现都可以在运行的过程中去创建和选择使用哪种。出于后向兼容的目的，这些实现可以通过所需的API级别而有所变化。一个实现可能会使用最新发布的APIs，而其他的则会去使用比较老的APIs。

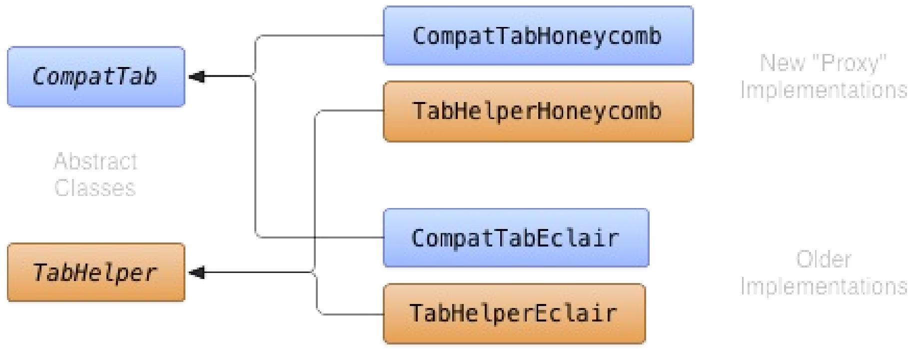
创建抽象的Tab接口

为了能够创建一个向后兼容的tabs，你首先需要决定你的应用需要哪些功能和哪些特定的APIs接口。在顶层分节tabs的情况下，假设你有以下功能需求：

1. 显示图标和文本的Tab指示器
2. Tabs可以跟一个Fragment实例向关联
3. Activity可以监听到Tab变化

提前准备这些需求能够让你控制抽象层的范围。这意味着你可以花更少的时间去创建抽象层的多个具体实现，并很快就能使用这些新的后向兼容的实现。

Tabs的关键APIs是ActionBar和ActionBar.Tab，为了能够使得tab能够感知Android版本，这些是需要抽象出来的APIs。这个示例项目的需求要求同Eclair(API等级5)保持一致性，同时能够利用Honeycomb(API等级11)中新的tab功能。一张展示能够支持这两种实现的类结构和它们的抽象父类的图显示如下：



- 图1.抽象基类和版本相关的子类实现类结构图

Abstract ActionBar.Tab

通过创建一个代表tab的抽象类来开始着手构建tab抽象层，这个类是Actionbar.Tab接口的镜像:

```
public abstract class CompatTab {
    ...
    public abstract CompatTab setText(int resId);
    public abstract CompatTab setIcon(int resId);
    public abstract CompatTab setTabListener(
            CompatTabListener callback);
    public abstract CompatTab setFragment(Fragment fragment);
    public abstract CharSequence getText();
    public abstract Drawable getIcon();
    public abstract CompatTabListener getCallback();
    public abstract Fragment getFragment();
    ...
}
```

在这里，为了简化诸如tab对象和Activity的联系（未在代码片段中显示）等公共的功能，你可以使用一个抽象类而不是去使用接口。

抽象出Action Bar Tab的方法

下一步，定义一个能够允许你往Activity中创建和添加tab抽象类，并定义类似ActionBar.newTab())和ActionBar.addTab())的方法。

```
public abstract class TabHelper {
    ...
    public CompatTab newTab(String tag) {
        // This method is implemented in a later lesson.
    }
    public abstract void addTab(CompatTab tab);
    ...
}
```

在下一课程中，你将会创建TabHelper和CompatTab的实现，它能够在新旧不同的平台版本上都能工作。

代理至新的APIs

> 编写: spencer198711 - 原文:http://developer.android.com/training/backward-compatible-ui/new-implementation.html

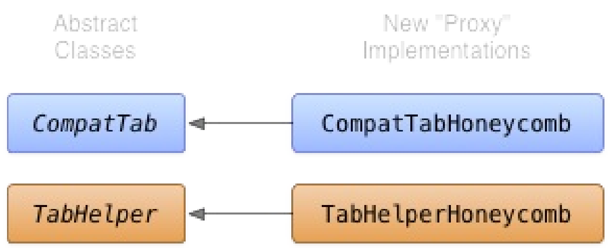
这一课展示了如何编写CompatTab和TabHelper等抽象类的子类，并且使用了较新的APIs。你的应用可以在支持这些新的APIs的平台版本的设备上使用这种实现方式。

使用较新的APIs实现Tabs

CompatTab和TabHelper抽象类的具体子类是一种代理实现，它们使用了使用较新的APIs。由于抽象类在之前的课程中定义并且是对新APIs接口（类结构、方法签名等等）的镜像，使用新APIs的具体子类只是简单的代理方法调用和方法调用的结果。

你可以在这些具体子类中直接使用较新的APIs，由于使用延迟类加载的方式，在早期版本的设备上并不会发生崩溃现象。这些类在首次次被访问（实例化类对象或者访问类的静态属性或静态方法）的时候才会去加载并初始化。因此，只要你不在Honeycomb之前的设备上实例化Honeycomb相关的实现，dalvik虚拟机都不会抛出VerifyError异常。

对于本实现，一个比较好的命名约定是把具体子类需要的API等级或者版本名字附加在APIs接口的后边。例如，本地tab实现可以由 `CompatTabHoneycomb` 和 `abHelperHoneycomb` 这两个类提供，名字后面附加Honeycomb是由于它们都依赖于Android 3.0（API等级11）之后版本的APIs。



- 图1. Honeycomb上tabs实现的类关系图.

实现CompatTabHoneycomb

`CompatTabHoneycomb` 是 `CompatTab` 抽象类的具体实现并用来引用单独的tabs。 `CompatTabHoneycomb` 只是简单的代理ActionBar.Tab对象的方法调用。 开始使用ActionBar.Tab的APIs实现CompatTabHoneycomb：

```
public class CompatTabHoneycomb extends CompatTab {
    // The native tab object that this CompatTab acts as a proxy for.
    ActionBar.Tab mTab;
    ...
    protected CompatTabHoneycomb(FragmentActivity activity, String tag) {
        ...
        // Proxy to new ActionBar.newTab API
        mTab = activity.getActionBar().newTab();
    }
    public CompatTab setText(int resId) {
        // Proxy to new ActionBar.Tab.setText API
        mTab.setText(resId);
        return this;
    }
    ...
    // Do the same for other properties (icon, callback, etc.)
}
```

实现TabHelperHoneycomb

`TabHelperHoneycomb` 是 `TabHelper` 抽象类的具体实现，`TabHelperHoneycomb` 代理方法调用到ActionBar对象，而这个ActionBar对象是从包含他的Activity中获取的。

实现 `TabHelperHoneycomb` ，代理其方法调用到ActionBar的API：

```
public class TabHelperHoneycomb extends TabHelper {
    ActionBar mActionBar;
    ...

    protected void setUp() {
        if (mActionBar == null) {
            mActionBar = mActivity.getActionBar();
            mActionBar.setNavigationMode(
                    ActionBar.NAVIGATION_MODE_TABS);
        }
    }

    public void addTab(CompatTab tab) {
        ...
        // Tab is a CompatTabHoneycomb instance, so its
        // native tab object is an ActionBar.Tab.
        mActionBar.addTab((ActionBar.Tab) tab.getTab());
    }

    // The other important method, newTab() is part of
    // the base implementation.
}
```

使用旧的APIs实现新API的效果

> 编写: spencer198711 - 原文:http://developer.android.com/training/backward-compatible-ui/older-implementation.html

这一课讨论了如何创建一个支持旧的设备并且与新的APIs接口相同的实现。

决定一个替代方案

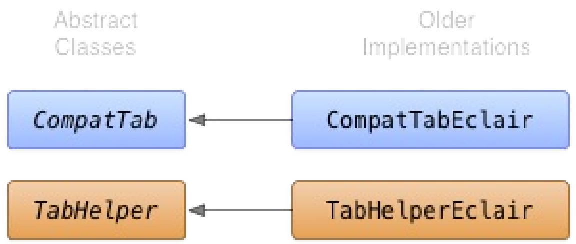
在以向后兼容的方式使用较新的UI功能的时候，最具挑战的任务是为旧的平台版本决定一个解决方案。在很多情况下，使用旧的UI框架中的功能是有可能完成这些新的UI组件的。例如：

- Action Bar可以使用水平的包含图片按钮的LinearLayout来实现，这个在Activity中的LinearLayout作为自定义标题栏或者仅仅作为视图。下拉功能行为可以使用设备的菜单按钮来实现。
- Action Bar的tab页可以使用包含按钮的水平的LinearLayout，或者使用TabWidgetUI控件来实现。
- NumberPicker和Switch控件可以分别通过使用Spinner和ToggleButton控件来实现。
- ListPopupWindow和PopupMenu控件可以通过使用PopupWindow来实现。

为了往老的设备上向后移植UI组件，这些一般不是一刀切的解决方案。注意用户体验：在老的设备上，用户可能不熟悉新的界面设计模式和UI组件，思考一下如何使用熟悉的控件去实现相同的功能。在很多种情况下，这些通常不会被注意到，特别是在如果新的UI组件在应用程序的生态系统中是突出的（比如Action Bar），或者交互模型是非常简单和直观的(比如使用ViewPager去滑动界面)。

使用旧的APIs实现Tabs

你可以使用TabWidget和TabHost（尽管其中一个也可以使用水平方向的Button控件）去创建Action Bar Tabs的老的实现。可以在TabHelperEclair和CompatTabEclair的类中去实现，因为这些实现使用了不迟于Android 2.0（Eclair）的APIs。



- 图1. Eclair版本上实现tabs的类图

`CompatTabEclair` 在实例变量中保存了诸如tab文本和tab图标等tab属性，因为在老的版本中没有ActionBar.Tab对象去处理这些数据存储。

```
public class CompatTabEclair extends CompatTab {
    // Store these properties in the instance,
    // as there is no ActionBar.Tab object.
    private CharSequence mText;
    ...

    public CompatTab setText(int resId) {
        // Our older implementation simply stores this
        // information in the object instance.
        mText = mActivity.getResources().getText(resId);
        return this;
    }

    ...
    // Do the same for other properties (icon, callback, etc.)
}
```

`TabHelperEclair` 利用了TabHost控件的方法去创建TabHost.TabSpec对象和tab的页面指示效果：

```
public class TabHelperEclair extends TabHelper {
    private TabHost mTabHost;
    ...

    protected void setUp() {
        if (mTabHost == null) {
            // Our activity layout for pre-Honeycomb devices
            // must contain a TabHost.
            mTabHost = (TabHost) mActivity.findViewById(
                    android.R.id.tabhost);
            mTabHost.setup();
        }
    }

    public void addTab(CompatTab tab) {
        ...
        TabSpec spec = mTabHost
                .newTabSpec(tag)
                .setIndicator(tab.getText()); // And optional icon
        ...
        mTabHost.addTab(spec);
    }

    // The other important method, newTab() is part of
    // the base implementation.
}
```

现在你已经有了两种 `CompatTab` 和 `TabHelper` 的实现，一种是使用了新的APIs为了能够在Android 3.0或其后版本设备上能够运行，另一种则是使用了旧的APIs为了在Android 2.0或之前的设备上能够运行。下一课讨论在应用中使用这两种实现。

使用能感知版本的组件

> 编写:spencer198711 - 原文:http://developer.android.com/training/backward-compatible-ui/using-component.html

既然对 `TabHelper` 和 `CompatTab` 你已经有了两种具体实现，一个为Android 3.0和其后版本，一个为Android 3.0之前的版本。现在，该使用这些实现做些事情了。这一课讨论了创建在这两种实现之前切换的逻辑，创建能够感知版本的界面布局，最终使用我们创建的后向兼容的UI组件。

添加切换逻辑

`TabHelper` 抽象类基于当前设备的平台版本，是用来创建适当版本的 `TabHelper` 和 `CompatTab` 实例的工厂类：

```
public abstract class TabHelper {
    ...
    // Usage is TabHelper.createInstance(activity)
    public static TabHelper createInstance(FragmentActivity activity) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new TabHelperHoneycomb(activity);
        } else {
            return new TabHelperEclair(activity);
        }
    }

    // Usage is mTabHelper.newTab("tag")
    public CompatTab newTab(String tag) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new CompatTabHoneycomb(mActivity, tag);
        } else {
            return new CompatTabEclair(mActivity, tag);
        }
    }
    ...
}
```

创建能感知版本的Activity布局

下一步是提供能够支持两种tab实现的Activity界面布局。对于老的实现（TabHelperEclair），你需要确保你的界面布局包含TabWidget和TabHost，同时存在一个包含tab内容的布局容器。

res/layout/main.xml:

```
<!-- This layout is for API level 5-10 only. -->
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp">

        <TabWidget
            android:id="@android:id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

    </LinearLayout>
</TabHost>
```

对于 `TabHelperHoneycomb` 的实现，你唯一要做的就是一个包含tab内容的FrameLayout，这是由于ActionBar已经提供了tab相关的页面。

res/layout-v11/main.xml:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabcontent"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

在运行的时候，Android将会根据平台版本去决定使用哪个版本的 `main.xml` 布局文件。这根上一节中选择哪一个版本的 `TabHelper` 所展示的逻辑是相同的。

在Activity中使用TabHelper

在Activity的onCreate()方法中，你可以获得一个 `TabHelper` 对象，并且使用以下代码添加tabs：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    setContentView(R.layout.main);

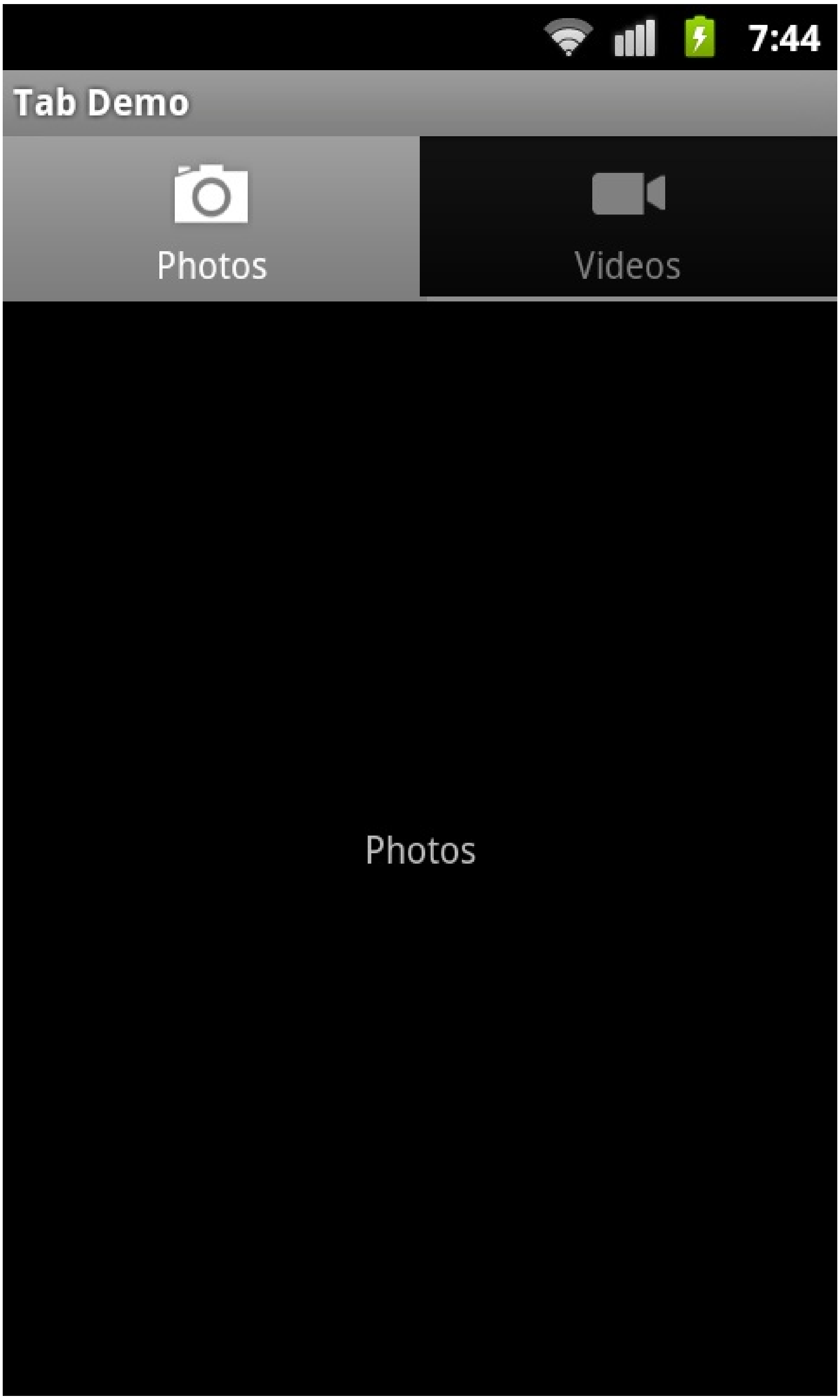
    TabHelper tabHelper = TabHelper.createInstance(this);
    tabHelper.setUp();

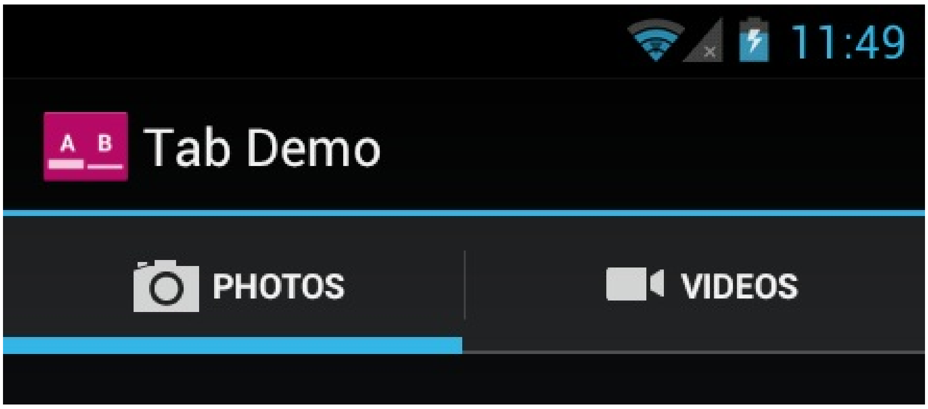
    CompatTab photosTab = tabHelper
            .newTab("photos")
            .setText(R.string.tab_photos);
    tabHelper.addTab(photosTab);

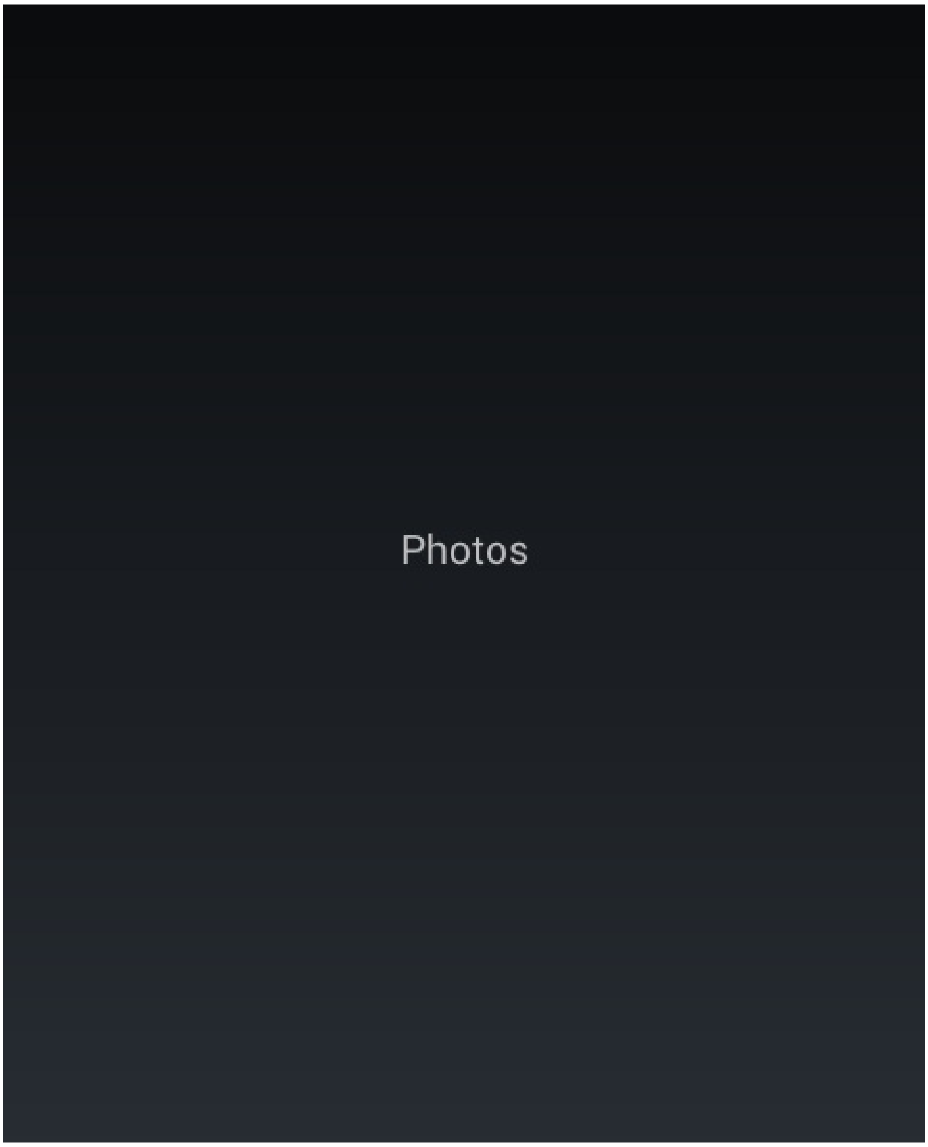
    CompatTab videosTab = tabHelper
            .newTab("videos")
            .setText(R.string.tab_videos);
    tabHelper.addTab(videosTab);
}
```

当运行这个应用的时候，代码会自动显示对应的界面布局和实例化对应的 `TabHelperHoneycomb` 或 `TabHelperEclair` 对象，而实际使用的类对于Actvity来说是不透明的，因为它们拥有共同的 `TabHelper` 接口。

以下是这种实现运行在Android 2.3和Android 4.0上的界面截图：


7:44
Tab Demo
Photos
Videos
Photos


11:49
Tab Demo
PHOTOS
VIDEOS



- 图1.向后兼容的tabs运行在Android 2.3设备上（使用TabHelperEclair）和运行在Android 4.0设备上的截图

实现辅助功能

> 编写:K0ST - 原文:http://developer.android.com/training/accessibility/index.html

当我们需要尽可能扩大我们用户的基数的时候，就要开始注意我们软件的可达性了(*Accessibility* 易接近，可亲性)。在界面中展示提示对大多数用户而言是可行的，比如说当按钮被按下时视觉上的变化，但是对于那些视力上有些缺陷的用户而言效果就不是那么理想了。

本章将给您演示如何最大化利用Android框架中的Accessibility特性。包括如何利用焦点导航(*focus navigation*)与内容描述(*content description*)对你的应用的可达性进行优化。也包括了创建Accessibility Service， 使用户与应用（不仅仅是你自己的应用）之间的交互更加容易。

Lessons

- 开发**Accessibility**应用

  学习如何让你的程序更易用，具有可达性。 允许使用键盘或者十字键(*directional pad*)来进行导航，利用Accessibility Service特性设置标签或执行事件来打造更舒适的用户体验。

- 编写 **Accessibility Services**

  编写一个Accessibility Service来监听可达性事件，利用这些不同类型的事件和内容描述来帮助用户与应用的交互。本例将会实现利用一个TTS引擎来向用户发出语音提示的功能。

开发辅助程序

> 编写:K0ST - 原文:http://developer.android.com/training/accessibility/accessible-app.html

本课程将教您：

1. 添加内容描述(*Content Descriptions*)
2. 设计焦点导航（*Focus Navigation*）
3. 触发可达性事件(*Accessibility Events*)
4. 测试你的程序

Android平台本身有一些专注可达性的特性，这些特性可以帮助你专门为那些视觉上或生理上有缺陷的用户在应用上做特别的优化。然而，正确的优化方式或最简单利用这个特性的方法往往不是那么显而易见的。本课程将给您演示如何利用和实现这些策略和平台的特性功能，构建一个更友好的具有可达性的Android应用。

添加内容描述

一个好的交互界面上的元素通常不需要特别使用一个标签来表明这个元素的作用。例如对于一个任务型应用来说，一个项目旁边的勾选框表达的意思就非常明确，或者对于一个文件管理应用，垃圾桶的图标表达的意思也非常清除。然而对于具有视觉障碍的用户来说，其他类型的UI交互提示是有必要的。

幸运的是，我们可以很轻松的给一个UI元素加上标签，这样类似于TalkBack这样的基于语音的Accessibility Service就可以将标签的内容朗读出来。如果你的标签在整个应用的生命周期中不太可能会发生变化(比如‘停止’或者‘购买’)，你就可以在XML布局文件中对*android:contentDescription*属性进行设置。代码如下：

```
<Button
    android:id="@+id/pause_button"
    android:src="@drawable/pause"
    android:contentDescription="@string/pause"/>
```

然而，在很多情况下描述的内容是基于上下文环境的，比如说一个开关按钮的状态，或者在list中一片可选的数据项。在运行时编辑内容描述可以使用*setContentDescription()*方法，代码如下：

```
String contentDescription = "Select " + strValues[position];
label.setContentDescription(contentDescription);
```

将以上功能添加进您的代码是提高您应用可达性的最简单的方法。尝试着将那些有用的地方都加入内容描述，但同时要避免像web开发者那样将所有的元素都标注，那样会产生大量的无用信息。比如说，不要将应用图标的内容描述设置为‘应用图标’。这只会对用户的浏览产生干扰。

来试试吧！下载TalkBack(谷歌开发的一款可达性应用)，在**Settings > Accessibility > TalkBack**将它开启。然后使用你的应用听听看TalkBack发出的语音提示。

设计焦点导航

你的应用除了支持触摸操作外，更应该支持其他的导航方式。很多Android设备不仅仅提供了触摸屏，还提供了其他的导航硬件比如说十字键、方向键、轨迹球等等。除此之外，最新的Android发行版本也支持蓝牙或USB的外接设备，比如键盘等等。

为了实现这种方式的导航，一切用户可以用来可导航的元素(*navigational elements*)都需要设置为focusable（聚焦）,它可以在运行时通过*View.setFocusable()*方法来进行设定，或者也可以在XML布局文件中使用*android:focusable*来设置。

每个UI控件有四个属性，*android:nextFocusUp*,*android:nextFocusDown*,*android:nextFocusLeft*,*android:nextFocusRight*,用户在导航时可以利用这些属性来指定下一个焦点的位置。系统会自动根据布局的方向来确定导航的顺序，如果在您的应用中系统提供的方案并不合适，您可以用这些属性来进行自定义的修改。

比如说，下面就是一个关于按钮和标签的例子，他们都是可聚焦的(*focusable*)，按向下键会将焦点从按钮移到文字上，按向上会重新将焦点移到按钮上。

```
<Button android:id="@+id/doSomething"
    android:focusable="true"
    android:nextFocusDown=”@id/label”
    ... />
<TextView android:id="@+id/label"
    android:focusable=”true”
    android:text="@string/labelText"
    android:nextFocusUp=”@id/doSomething”
    ... />
```

证实您的应用运行正确的直观方法，最简单的方式就是在Android虚拟机里运行您的应用，然后使用虚拟器的方向键来在各个元素之间导航，使用OK按钮来代替触摸操作。

触发可达性事件

如果你在你的Android框架中使用了View组件，当你选中了一个View或者是焦点变化的时候，可达性事件(*AccessibilityEvent*)都会产生。这些事件会被传递到Accessibility Service中进行处理，实现一些辅助功能，如语音提示等。

如果你写了一个自定义的View，请确保它在合适的时候产生事件。使用*sendAccessibilityEvent(int)*函数可以产生可达性事件，其中的参数表示事件的类型。完整的可达性事件类型可查阅AccessibilityEvent参考文档。

比如说，你拓展了一个图片的View，你希望在它聚焦的时候使用键盘打字可以在其中插入题注，这时候发送一个*TYPE_VIEW_TEXT_CHANGED*事件就非常合适，尽管它不是本身就构建在这个图片View中的。产生事件的代码如下：

```
public void onTextChanged(String before, String after) {
    ...
    if (AccessibilityManager.getInstance(mContext).isEnabled()) {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);
    }
    ...
}
```

测试你的程序

请确保您在添加可达性功能后测试它的有效性。为了测试内容描述可达性事件，请安装并启用一个Accessibility Service。比如说使用TalkBack，它是一个免费的开源的屏幕读取软件，可在Google Play上进行下载。Service启动后，请测试您应用中所有的功能，同时听听TalkBack的语音反馈。

同时，尝试着用一个方向控制器来控制你的应用，而非使用直接触摸的方式。你可以使用一个物理设备，比如十字键、轨迹球等。如果没有条件，可以使用android虚拟器，它提供了虚拟的按键控制。

在测试导航与反馈的同时，和在没有任何视觉提示的情况下，应该对你的应用大概是一个什么样子有所认识。出现问题就修复优化它们，最终就会开发出一个更易用可达的Android程序。

开发辅助服务

> 编写:KOST - 原文:http://developer.android.com/training/accessibility/service.html

本课程将教您：

1. 创建可达性服务(Accessibility Service)
2. 配置可达性服务(Accessibility Service)
3. 响应可达性事件(AccessibilityEvents)
4. 从View层级中提取更多信息

Accessibility Service是Android系统框架提供给安装在设备上应用的一个可选的导航反馈特性。Accessibility Service 可以替代应用与用户交流反馈，比如将文本转化为语音提示，或是用户的手指悬停在屏幕上一个较重要的区域时的触摸反馈等。本课程将教您如何创建一个Accessibility Service，同时处理来自应用的信息，并将这些信息反馈给用户。

创建Accessibility Service

Accessibility Service可以绑定在一个正常的应用中，或者是单独的一个Android项目都可以。创建一个Accessibility Service的步骤与创建普通Service的步骤相似，在你的项目中创建一个继承于AccessibilityService的类：

```
package com.example.android.apis.accessibility;

import android.accessibilityservice.AccessibilityService;

public class MyAccessibilityService extends AccessibilityService {
...
    @Override
    public void onAccessibilityEvent(AccessibilityEvent event) {
    }

    @Override
    public void onInterrupt() {
    }

...
}
```

与其他Service类似，你必须在manifest文件当中声明这个Service。记得标明它监听处理了 `android.accessibilityservice` 事件，以便Service在其他应用产生AccessibilityEvent的时候被调用。

```
<application ...>
...
<service android:name=".MyAccessibilityService">
     <intent-filter>
         <action android:name="android.accessibilityservice.AccessibilityService" />
     </intent-filter>
     . . .
</service>
...
</application>
```

如果你为这个Service创建了一个新项目，且仅仅是一个Service而不准备做成一个应用，那么你就可以移除启动的Activity(一般为MainActivity.java)，同样也记得在manifest中将这个Activity声明移除。

配置Accessibility Service

设置Accessibility Service的配置变量会告诉系统如何让Service运行与何时运行。你希望响应哪种类型的事件？Service是否对所有的应用有效还是对部分指定包名的应用有效？使用哪些不同类型的反馈？

你有两种设置这些变量属性的方法，一种向下兼容的办法是通过代码来进行设定，使用 `setServiceInfo` (android.accessibilityservice.AccessibilityServiceInfo)。你需要重写(*override*) `onServiceConnected()` 方法，并在这里进行Service的配置。

```
@Override
public void onServiceConnected() {
    // Set the type of events that this service wants to listen to.  Others
    // won't be passed to this service.
    info.eventTypes = AccessibilityEvent.TYPE_VIEW_CLICKED |
            AccessibilityEvent.TYPE_VIEW_FOCUSED;

    // If you only want this service to work with specific applications, set their
    // package names here.  Otherwise, when the service is activated, it will listen
    // to events from all applications.
    info.packageNames = new String[]
            {"com.example.android.myFirstApp", "com.example.android.mySecondApp"};

    // Set the type of feedback your service will provide.
    info.feedbackType = AccessibilityServiceInfo.FEEDBACK_SPOKEN;

    // Default services are invoked only if no package-specific ones are present
    // for the type of AccessibilityEvent generated.  This service *is*
    // application-specific, so the flag isn't necessary.  If this was a
    // general-purpose service, it would be worth considering setting the
    // DEFAULT flag.

    // info.flags = AccessibilityServiceInfo.DEFAULT;

    info.notificationTimeout = 100;

    this.setServiceInfo(info);

}
```

在Android 4.0之后，就用另一种方式来设置了：通过设置XML文件来进行配置。一些特性的选项比如 `canRetrieveWindowContent` 仅仅可以在XML可以配置。对于上面所示的相应的配置，利用XML配置如下：

```
<accessibility-service
     android:accessibilityEventTypes="typeViewClicked|typeViewFocused"
     android:packageNames="com.example.android.myFirstApp, com.example.android.mySecon
dApp"
     android:accessibilityFeedbackType="feedbackSpoken"
     android:notificationTimeout="100"
     android:settingsActivity="com.example.android.apis.accessibility.TestBackActivity"

     android:canRetrieveWindowContent="true"
/>
```

如果你确定是通过XML进行配置，那么请确保在manifest文件中通过< meta-data >标签指定这个配置文件。假设此配置文件存放的地址为： `res/xml/serviceconfig.xml` ，那么标签应该如下:

```
<service android:name=".MyAccessibilityService">
     <intent-filter>
         <action android:name="android.accessibilityservice.AccessibilityService" />
     </intent-filter>
     <meta-data android:name="android.accessibilityservice"
     android:resource="@xml/serviceconfig" />
</service>
```

响应Accessibility Event

现在你的Service已经配置好并可以监听Accessibility Event了，来写一些响应这些事件的代码吧！首先就是要重写*onAccessibilityEvent(AccessibilityEvent)*方法，在这个方法中，使用 `getEventType()` 来确定事件的类型，使用 `getContentDescription()` 来提取产生事件的View的相关的文本标签。

```
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
    final int eventType = event.getEventType();
    String eventText = null;
    switch(eventType) {
        case AccessibilityEvent.TYPE_VIEW_CLICKED:
            eventText = "Focused: ";
            break;
        case AccessibilityEvent.TYPE_VIEW_FOCUSED:
            eventText = "Focused: ";
            break;
    }

    eventText = eventText + event.getContentDescription();

    // Do something nifty with this text, like speak the composed string
    // back to the user.
    speakToUser(eventText);
    ...
}
```

从View层级中提取更多信息

这一步并不是必要步骤，但是却非常有用。Android 4.0版本中增加了一个新特性，就是能够用AccessibilityService来遍历View层级，并从产生Accessibility 事件的组件与它的父子组件中提取必要的信息。为了实现这个目的，你需要在XML文件中进行如下的配置：

```
android:canRetrieveWindowContent="true"
```

一旦完成，使用getSource())获取一个AccessibilityNodeInfo对象，如果触发事件的窗口是活动窗口，该调用只返回一个对象，如果不是,它将返回null，做出相应的反响。下面的示例是一个代码片段,当它接收到一个事件时,执行以下步骤:

1. 立即获取到产生这个事件的Parent
2. 在这个Parent中寻找文本标签或勾选框
3. 如果找到，创建一个文本内容来反馈给用户，提示内容和是否已勾选。
4. 如果当遍历View的时候某处返回了null值，那么就直接结束这个方法。

```
// Alternative onAccessibilityEvent, that uses AccessibilityNodeInfo

@Override
public void onAccessibilityEvent(AccessibilityEvent event) {

    AccessibilityNodeInfo source = event.getSource();
    if (source == null) {
        return;
    }

    // Grab the parent of the view that fired the event.
    AccessibilityNodeInfo rowNode = getListItemNodeInfo(source);
    if (rowNode == null) {
        return;
    }

    // Using this parent, get references to both child nodes, the label and the checkb
ox.
    AccessibilityNodeInfo labelNode = rowNode.getChild(0);
    if (labelNode == null) {
        rowNode.recycle();
        return;
    }

    AccessibilityNodeInfo completeNode = rowNode.getChild(1);
    if (completeNode == null) {
        rowNode.recycle();
        return;
    }

    // Determine what the task is and whether or not it's complete, based on
    // the text inside the label, and the state of the check-box.
    if (rowNode.getChildCount() < 2 || !rowNode.getChild(1).isCheckable()) {
        rowNode.recycle();
        return;
    }

    CharSequence taskLabel = labelNode.getText();
    final boolean isComplete = completeNode.isChecked();
    String completeStr = null;

    if (isComplete) {
        completeStr = getString(R.string.checked);
    } else {
        completeStr = getString(R.string.not_checked);
    }
    String reportStr = taskLabel + completeStr;
    speakToUser(reportStr);
}
```

现在你已经实现了一个完整可运行的Accessibility Service。尝试着调整它与用户的交互方式吧！比如添加语音引擎，或者添加震动来提供触觉上的反馈都是不错的选择！

管理系统UI

> 编写:KOST - 原文:http://developer.android.com/training/system-ui/index.html

System Bar是用来展示通知、表现设备状态和完成设备导航的屏幕区域。通常上来说，系统栏(System bar)包括状态栏和导航栏(Figure 1)，他们一般都是与程序同时显示在屏幕上的。而照片、视频等这类沉浸式的应用可以临时弱化系统栏图标来创造一个更加专注的体验环境，甚至可以完全隐藏系统Bar。



Figure 1. System bars，包含[1]状态栏，和[2]导航栏。

如果你对Android Design Guide很熟悉，你应该已经知道遵照标准的Android UI Guideline与遵循模式来设计App的重要性。在你修改系统栏之前，你应该仔细的考虑一下用户的需求与预期，因为它们是操作设备和观察设备状态的的常规途径。

这节课描述了如何在不同版本的Android上隐藏或淡化系统栏，来营造一个沉浸式的用户体验，同时做到快速的访问与操作系统栏。

Sample

ImmersiveMode - http://developer.android.com/samples/ImmersiveMode/index.html

Lessons

- 淡化系统栏

  学习如何淡化和隐藏状态栏与导航栏。

- 隐藏状态栏

  学习如何在不同版本的Android上隐藏状态栏。

- 隐藏导航栏

  学习如何隐藏导航栏。

- 全屏沉浸式应用

  学习如何在你的App中创建沉浸模式。

- 响应**UI**可见性的变化

  学习如何注册一个监听器来监听系统UI可见性的变化，以便于相应的调整App的UI。

淡化系统Bar

> 编写:K0ST - 原文:http://developer.android.com/training/system-ui/dim.html

本课程将向你讲解如何在Android 4.0(*API level 14*)与更高的的系统版本上淡化系统栏(System bar,状态栏与导航栏)。早期版本的Android没有提供一个自带的方法来淡化系统栏。

当你使用这个方法的时候，内容区域并不会发生大小的变化，只是系统栏的图标会收起来。一旦用户触摸状态栏或者是导航栏的时候，这两个系统栏就又都会完全显示（无透明度）。这种方法的优势是系统栏仍然可见，但是它们的细节被隐藏掉了，因此可以在不牺牲快捷访问系统栏的情况下创建一个沉浸式的体验。

这节课将教您

1. 淡化状态栏和导航栏
2. 显示状态栏和导航栏

同时您应该阅读

- Action Bar API 指南
- Android Design Guide

淡化状态栏和系统栏

如果要淡化状态和通知栏，在版本为4.0以上的Android系统上，你可以像如下使用 `SYSTEM_UI_FLAG_LOW_PROFILE` 这个标签。

```
// This example uses decor view, but you can use any visible view.
View decorView = getActivity().getWindow().getDecorView();
int uiOptions = View.SYSTEM_UI_FLAG_LOW_PROFILE;
decorView.setSystemUiVisibility(uiOptions);
```

一旦用户触摸到了状态栏或者是系统栏，这个标签就会被清除，使系统栏重新显现（无透明度）。在标签被清除的情况下，如果你想重新淡化系统栏就必须重新设定这个标签。

图1展示了一个图库中的图片，界面的系统栏都已被淡化（需要注意的是图库应用完全隐藏状态栏，而不是淡化它）；注意导航栏（图片的右侧）上变暗的白色的小点，他们代表了被隐藏的导航操作。



图**1.**淡化的系统栏

图2展示的是同一张图片，系统栏处于显示的状态。

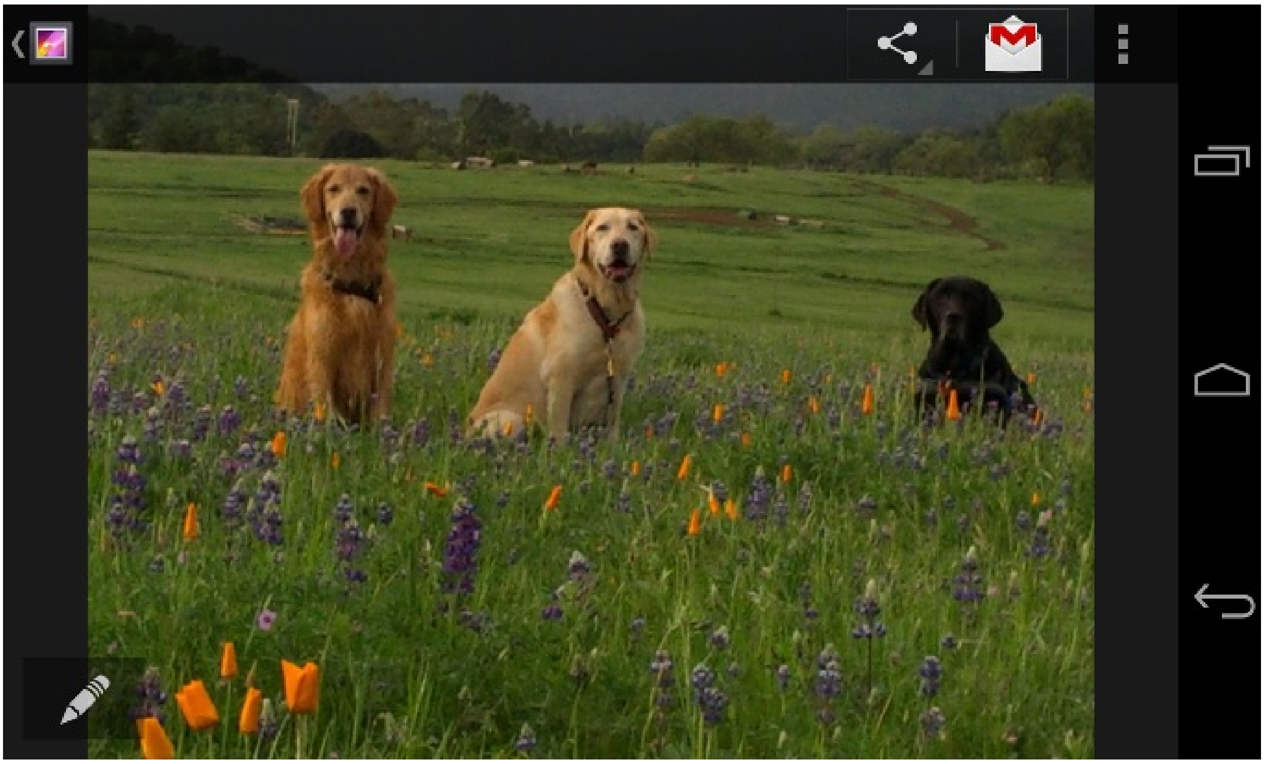


图**2.**显示的系统栏

显示状态栏与导航栏

如果你想动态的清除显示标签，你可以使用 `setSystemUiVisibility()` 方法：

```
View decorView = getActivity().getWindow().getDecorView();
// Calling setSystemUiVisibility() with a value of 0 clears
// all flags.
decorView.setSystemUiVisibility(0);
```

隐藏状态栏

> 编写:KOST - 原文:http://developer.android.com/training/system-ui/status.html

这节课将教您

1. 在4.0及以下版本中隐藏状态栏
2. 在4.1及以上版本中隐藏状态栏
3. 在4.4及以上版本中隐藏状态栏
4. 让内容显示在状态栏之后
5. 同步状态栏与Action Bar的变化

同时您应该阅读

- Action Bar API 指南
- Android Design Guide

本课程将教您如何在不同版本的Android下隐藏状态栏。隐藏状态栏（或者是导航栏）可以让内容得到更多的展示空间，从而提供一个更加沉浸式的用户体验。

图1展示了显示状态栏的界面



图**1**. 显示状态栏.

图2展示了隐藏状态栏的界面。请注意，Action Bar这个时候也被隐藏了。请永远不要在隐藏状态栏的时候显示Action Bar。



图2. 隐藏状态栏.

在4.0及以下版本中隐藏状态栏

在Android 4.0及更低的版本中，你可以通过设置 `WindowManager` 来隐藏状态栏。你可以动态的隐藏，也可以在你的manifest文件中设置Activity的主题。如果你的应用的状态栏在运行过程中会一直隐藏，那么推荐你使用改写manifest设定主题的方法（严格上来讲，即便设置了manifest你也可以动态的改变界面主题）。

```
<application
    ...
    android:theme="@android:style/Theme.Holo.NoActionBar.Fullscreen" >
    ...
</application>
```

设置主题的优势是：

- 易于维护，且不像动态设置标签那样容易出错
- 有更流畅的UI转换，因为在初始化你的Activity之前，系统已经得到了需要渲染UI的信息

另一方面我们可以选择使用 `WindowManager` 来动态隐藏状态栏。这个方法可以更简单的在用户与App进行交互式展示与隐藏状态栏。

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // If the Android version is lower than Jellybean, use this call to hide
        // the status bar.
        if (Build.VERSION.SDK_INT < 16) {
            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                    WindowManager.LayoutParams.FLAG_FULLSCREEN);
        }
        setContentView(R.layout.activity_main);
    }
    ...
}
```

当你设置 `WindowManager` 标签之后（无论是通过Activity主题还是动态设置），这个标签都会一直生效直到你清除它。

设置了 `FLAG_LAYOUT_IN_SCREEN` 之后，你可以拥有与启用 `FLAG_FULLSCREEN` 后相同的屏幕区域。这个方法防止了状态栏隐藏和展示的时候内容区域的大小变化。

在4.1及以上版本中隐藏状态栏

在Android 4.1(API level 16)以及更高的版本中，你可以使用setSystemUiVisibility()来进行动态隐藏。 `setSystemUiVisibility()` 在View层面设置了UI的标签，然后这些设置被整合到了Window层面。 `setSystemUiVisibility()` 给了你一个比设置 `WindowManager` 标签更加粒度化的操作。下面这段代码隐藏了状态栏：

```
View decorView = getWindow().getDecorView();
// Hide the status bar.
int uiOptions = View.SYSTEM_UI_FLAG_FULLSCREEN;
decorView.setSystemUiVisibility(uiOptions);
// Remember that you should never show the action bar if the
// status bar is hidden, so hide that too if necessary.
ActionBar actionBar = getActionBar();
actionBar.hide();
```

注意以下几点：

- 一旦UI标签被清除(比如跳转到另一个Activity),如果你还想隐藏状态栏你就必须再次设定它。详细可以看第五节如何监听并响应UI可见性的变化。
- 在不同的地方设置UI标签是有所区别的。如果你在Activity的onCreate()方法中隐藏系统栏，当用户按下home键系统栏就会重新显示。当用户再重新打开Activity的时候，onCreate()不会被调用，所以系统栏还会保持可见。如果你想让在不同Activity之间切换

时，系统UI保持不变，你需要在onResume()与onWindowFocusChaned()里设定UI标签。

- setSystemUiVisibility()仅仅在被调用的View显示的时候才会生效。
- 当从View导航到别的地方时，用setSystemUiVisibility()设置的标签会被清除。

让内容显示在状态栏之后

在Android 4.1及以上版本，你可以将应用的内容显示在状态栏之后，这样当状态栏显示与隐藏的时候，内容区域的大小就不会发生变化。要做到这个效果，我们需要用到 `SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN` 这个标志。同时，你也有可能需要 `SYSTEM_UI_FLAG_LAYOUT_STABLE` 这个标志来帮助你的应用维持一个稳定的布局。

当使用这种方法的时候，你就需要来确保应用中特定区域不会被系统栏掩盖（比如地图应用中一些自带的操作区域）。如果被覆盖了，应用可能就会无法使用。在大多数的情况下，你可以在布局文件中添加 `android:fitsSystemWindows` 标签，设置它为true。它会调整父ViewGroup使它留出特定区域给系统栏，对于大多数应用这种方法就足够了。

在一些情况下，你可能需要修改默认的padding大小来获取合适的布局。为了控制内容区域的布局相对系统栏（它占据了一个叫做“内容嵌入” `content insets` 的区域）的位置，你可以重写 `fitSystemWindows(Rect insets)` 方法。当窗口的内容嵌入区域发生变化时， `fitSystemWindows()` 方法会被view的hierarchy调用，让View做出相应的调整适应。重写这个方法你就可以按你的意愿处理嵌入区域与应用的布局。

同步状态栏与**Action Bar**的变化

在Android 4.1及以上的版本，为了防止在Action Bar隐藏和显示的时候布局发生变化，你可以使用Action Bar的overlay模式。在Overlay模式中，Activity的布局占据了所有可能的空间，好像Action Bar不存在一样，系统会在布局的上方绘制Aciton Bar。虽然这会遮盖住上方的一些布局，但是当Action Bar显示或者隐藏的时候，系统就不需要重新改变布局区域的大小，使之无缝的变化。

要启用Action Bar的overlay模式，你需要创建一个继承自Action Bar主题的自定义主题，将 `android:windowActionBarOverlay` 属性设置为true。要了解详细信息，请参考添加Action Bar课程中的Action Bar的覆盖层叠。

设置 `SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN` 来让你的activity使用的屏幕区域与设置 `SYSTEM_UI_FLAG_FULLSCREEN` 时的区域相同。当你需要隐藏系统UI时，使用 `SYSTEM_UI_FLAG_FULLSCREEN` 。这个操作也同时隐藏了Action Bar（因为 `windowActionBarOverlay="true"` ），当同时显示与隐藏ActionBar与状态栏的时候，使用一个动画来让他们相互协调。

隐藏导航栏

> 编写:[K0ST](#) - 原文:http://developer.android.com/training/system-ui/navigation.html

这节课将教您

1. 在4.0及以上版本中隐藏导航栏
2. 让内容显示在导航栏之后

本节课程将教您如何对导航栏进行隐藏，这个特性是Android 4.0（）版本中引入的。

即便本小节仅关注如何隐藏导航栏，但是在实际的开发中，你最好让状态栏与导航栏同时消失。在保证导航栏易于再次访问的情况下，隐藏导航栏与状态栏使内容区域占据了整个显示空间，因此可以提供一个更加沉浸式的用户体验。



图**1**. 导航栏.

在4.0及以上版本中隐藏导航栏

你可以在Android 4.0以及以上版本，使用 `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 标志来隐藏导航栏。这段代码同时隐藏了导航栏和系统栏：

```
View decorView = getWindow().getDecorView();
// Hide both the navigation bar and the status bar.
// SYSTEM_UI_FLAG_FULLSCREEN is only available on Android 4.1 and higher, but as
// a general rule, you should design your app to hide the status bar whenever you
// hide the navigation bar.
int uiOptions = View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
              | View.SYSTEM_UI_FLAG_FULLSCREEN;
decorView.setSystemUiVisibility(uiOptions);
```

注意以下几点

- 使用这个方法时，触摸屏幕的任何一个区域都会使导航栏（与状态栏）重新显示。用户

的交互会使这个标签`SYSTEM_UI_FLAG_HIDE_NAVIGATION`被清除。

- 一旦这个标签被清除了，如果你想再次隐藏导航栏，你就需要重新对这个标签进行设定。在下一节响应UI可见性的变化中，将详细讲解应用监听系统UI变化来做出相应的调整操作。
- 在不同的地方设置UI标签是有所区别的。如果你在Activity的onCreate()方法中隐藏系统栏，当用户按下home键系统栏就会重新显示。当用户再重新打开activity的时候，onCreate()不会被调用，所以系统栏还会保持可见。如果你想让在不同Activity之间切换时，系统UI保持不变，你需要在onReasume()与onWindowFocusChaned()里设定UI标签。
- setSystemUiVisibility()仅仅在被调用的View显示的时候才会生效。
- 当从View导航到别的地方时，用setSystemUiVisibility()设置的标签会被清除。

2)让内容显示在导航栏之后

在Android 4.1与更高的版本中，你可以让应用的内容显示在导航栏的后面，这样当导航栏展示或隐藏的时候内容区域就不会发生布局大小的变化。可以使用`SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION`标签来做到这个效果。同时，你也有可能需要`SYSTEM_UI_FLAG_LAYOUT_STABLE`这个标签来帮助你的应用维持一个稳定的布局。

当你使用这种方法的时候，就需要你来确保应用中特定区域不会被系统栏掩盖。更详细的信息可以浏览隐藏状态栏一节。

全屏沉浸式应用

> 编写:[K0ST](https://github.com/K0ST) - 原文:http://developer.android.com/training/system-ui/immersive.html

这节课将教您

1. 选择一种沉浸方式
2. 使用非粘性沉浸模式
3. 使用粘性沉浸模式

Adnroid 4.4(API level 19)中引入为 `setSystemUiVisibility()` 引入了一个新标签 `SYSTEM_UI_FLAG_IMMERSIVE` ，它可以让应用进入真正的全屏模式。当这个标签与 `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 和 `SYSTEM_UI_FLAG_FULLSCREEN` 一起使用的时候，导航栏和状态栏就会隐藏，让你的应用可以接受屏幕上任何地方的触摸事件。

当沉浸式全屏模式启用的时候，你的Activity会继续接受各类的触摸事件。用户可以通过在边缘区域向内滑动来让系统栏重新显示。这个操作清空了 `SYSTEM_UI_FLAG_HIDE_NAVIGATION` (和 `SYSTEM_UI_FLAG_FULLSCREEN` ，如果有的话)两个标签，因此系统栏重新变得可见。如果设置了的话，这个操作同时也触发了 `View.OnSystemUiVisibilityChangeListener` 。然而， 如果你想让系统栏在一段时间后自动隐藏的话，你应该使用 `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` 标签。请注意，带有'sticky'的标签不会触发任何的监听器，因为在这个模式下展示的系统栏是处于暂时(transient)的状态。

图1展示了各种不同的“沉浸式”状态

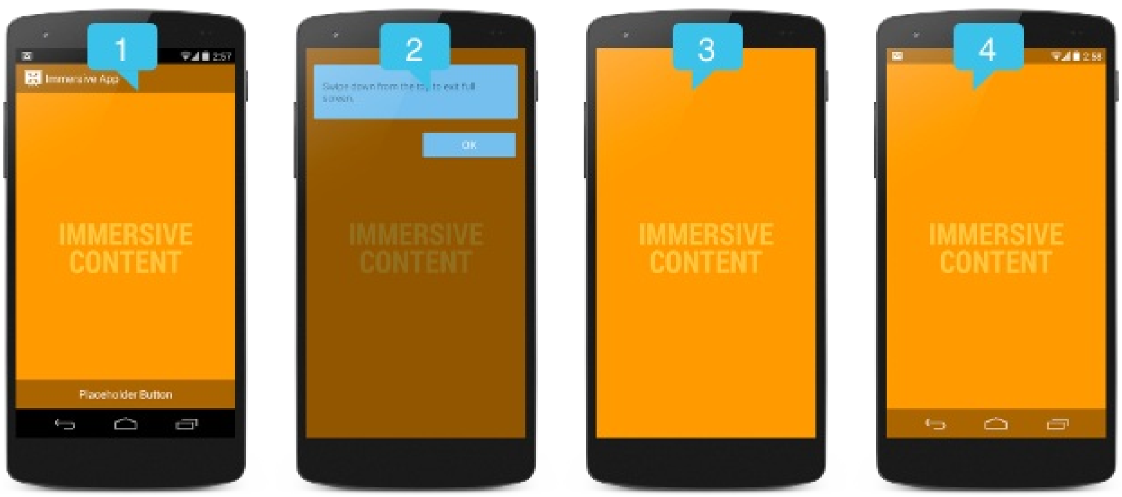


图**1**. 沉浸模式状态.

在上图中：

1. 非沉浸模式 —— 展示了应用进入沉浸模式之前的状态。也展示了设置 `IMMERSIVE` 标签后用户滑动展示系统栏的状态。用户滑动后， `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 和 `SYSTEM_UI_FLAG_FULLSCREEN` 就会被清除，系统栏就会重新显示并保持可见。 请注意，最好的实践方式就是让所有的UI控件的变化与系统栏的显示隐藏保持同步，这样可以减少屏幕显示所处的状态，同时提供了更无缝平滑的用户体验。因此所有的UI控件跟随系统栏一同显示。一旦应用进入了沉浸模式，相应的UI控件也跟随着系统栏一同隐藏。为了确保UI的可见性与系统栏保持一致，我们需要一个监听器 `View.OnSystemUiVisibilityChangeListener` 来监听系统栏的变化。这在下一节中将详细讲解。

2. 提示气泡——第一次进入沉浸模式时，系统将会显示一个提示气泡，提示用户如何再让系统栏显示出来。

   > **Note**：如果为了测试你想强制显示提示气泡，你可以先将应用设为沉浸模式，然后按下电源键进入锁屏模式，并在5秒中之后打开屏幕。

3. 沉浸模式—— 这张图展示了隐藏了系统栏和其他UI控件的状态。你可以设置 `IMMERSIVE` 和 `IMMERSIVE_STICKY` 来进入这个状态。

4. 粘性标签——这就是你设置了 `IMMERSIVE_STICKY` 标签时的UI状态，用户会向内滑动以展示系统栏。半透明的系统栏会临时的进行显示，一段时间后自动隐藏。滑动的操作并不会清空任何标签，也不会触发系统UI可见性的监听器，因为暂时显示的导航栏并不被认为是一种可见性状态的变化。

> **Note**： `immersive` 类的标签只有在与 `SYSTEM_UI_FLAG_HIDE_NAVIGATION` , `SYSTEM_UI_FLAG_FULLSCREEN` 中一个或两个一起使用的时候才会生效。你可以只使用其中的一个，但是一般情况下你需要同时隐藏状态栏和导航栏以达到沉浸的效果。

选择一种沉浸方式

`SYSTEM_UI_FLAG_IMMERSIVE` 与 `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` 都提供了沉浸式的体验，但是在上面的描述中，他们是不一样的，下面讲解一下什么时候该用哪一种标签。

- 如果你在写一款图书浏览器、新闻杂志阅读器，请将 `IMMERSIVE` 标签与 `SYSTEM_UI_FLAG_FULLSCREEN` , `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 一起使用。因为用户可能会经常访问Action Bar和一些UI控件，又不希望在翻页的时候有其他的东西进行干扰。 `IMMERSIVE` 在该种情况下就是个很好的选择。
- 如果你在打造一款真正的沉浸式应用，而且你希望屏幕边缘的区域也可以与用户进行交互，并且用户也不会经常访问系统UI。这个时候就要将 `IMMERSIVE_STICKY` 和 `SYSTEM_UI_FLAG_FULLSCREEN` `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 两个标签一起使用。比如做一款游戏或者绘图应用就很合适。

- 如果你在打造一款视频播放器，并且需要少量的用户交互操作。你可能就需要之前版本的一些方法了（从Android 4.0开始）。对于这种应用，简单的使用 `SYSTEM_UI_FLAG_FULLSCREEN` 与 `SYSTEM_UI_FLAG_HIDE_NAVIGATION` 就足够了，不需要使用 `immersive` 标签。

使用非粘性沉浸模式

当你使用 `SYSTEM_UI_FLAG_IMMERSIVE` 标签的时候，它是基于其他设置过的标签(`SYSTEM_UI_FLAG_HIDE_NAVIGATION` 和 `SYSTEM_UI_FLAG_FULLSCREEN`)来隐藏系统栏的。当用户向内滑动，系统栏重新显示并保持可见。

用其他的UI标签(如 `SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION` 和 `SYSTEM_UI_FLAG_LAYOUT_STABLE`)来防止系统栏隐藏时内容区域大小发生变化是一种很不错的方法。你也需要确保Action Bar和其他系统UI控件同时进行隐藏。下面这段代码展示了如何在不改变内容区域大小的情况下，隐藏与显示状态栏和导航栏。

```
// This snippet hides the system bars.
private void hideSystemUI() {
    // Set the IMMERSIVE flag.
    // Set the content to appear under the system bars so that the content
    // doesn't resize when the system bars hide and show.
    mDecorView.setSystemUiVisibility(
            View.SYSTEM_UI_FLAG_LAYOUT_STABLE
            | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
            | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
            | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION // hide nav bar
            | View.SYSTEM_UI_FLAG_FULLSCREEN // hide status bar
            | View.SYSTEM_UI_FLAG_IMMERSIVE);
}

// This snippet shows the system bars. It does this by removing all the flags
// except for the ones that make the content appear under the system bars.
private void showSystemUI() {
    mDecorView.setSystemUiVisibility(
            View.SYSTEM_UI_FLAG_LAYOUT_STABLE
            | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
            | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN);
}
```

你可能同时也希望在如下的几种情况下使用 `IMMERSIVE` 标签来提供更好的用户体验：

- 注册一个监听器来监听系统UI的变化。
- 实现 `onWindowFocusChanged()` 函数。如果窗口获取了焦点，你可能需要对系统栏进行隐藏。如果窗口失去了焦点，比如说弹出了一个对话框或菜单，你可能需要取消那些将要在 `Handler.postDelayed()` 或其他地方的隐藏操作。

- 实现一个 `GestureDetector` ，它监听了 `onSingleTapUp(MotionEvent)` 事件。可以使用户点击内容区域来切换系统栏的显示状态。单纯的点击监听可能不是最好的解决方案，因为当用户在屏幕上拖动手指的时候（假设点击的内容占据了整个屏幕），这个事件也会被触发。

更多关于此话题的讨论，可以观看这个视频 DevBytes: Android 4.4 Immersive Mode

使用粘性沉浸模式

当使用了 `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` 标签的时候，向内滑动的操作会让系统栏临时显示，并处于半透明的状态。此时没有标签会被清除，系统UI可见性监听器也不会被触发。如果用户没有进行操作，系统栏会在一段时间内自动隐藏。

图2展示了当使用 `IMMERSIVE_STICKY` 标签时，半透明的系统栏展示与又隐藏的状态。



图**2**. 自动隐藏系统栏.

下面是一段实现代码。一旦窗口获取了焦点，只要简单的设置 `IMMERSIVE_STICKY` 与上面讨论过的其他标签即可。

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
        super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        decorView.setSystemUiVisibility(
                View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_FULLSCREEN
                | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY);}
}
```

> **Notes**：如果你想实现`IMMERSIVE_STICKY`的自动隐藏效果，同时也需要展示你自己的UI控件。你只需要使用`IMMERSIVE`与`Handler.postDelayed()`或其他类似的东西，让它几秒后重新进入沉浸模式即可。

响应UI可见性的变化

> 编写:K0ST - 原文:http://developer.android.com/training/system-ui/visibility.html

本节课将教你如果注册监听器来监听系统UI可见性的变化。这个方法在将系统栏与你自己的UI控件进行同步操作时很有用。

注册监听器

为了获取系统UI可见性变化的通知，我们需要对View注册`View.OnSystemUiVisibilityChangeListener`监听器。通常上来说，这个View是用来控制导航的可见性的。

例如你可以添加如下代码在onCreate中

```
View decorView = getWindow().getDecorView();
decorView.setOnSystemUiVisibilityChangeListener
        (new View.OnSystemUiVisibilityChangeListener() {
    @Override
    public void onSystemUiVisibilityChange(int visibility) {
        // Note that system bars will only be "visible" if none of the
        // LOW_PROFILE, HIDE_NAVIGATION, or FULLSCREEN flags are set.
        if ((visibility & View.SYSTEM_UI_FLAG_FULLSCREEN) == 0) {
            // TODO: The system bars are visible. Make any desired
            // adjustments to your UI, such as showing the action bar or
            // other navigational controls.
        } else {
            // TODO: The system bars are NOT visible. Make any desired
            // adjustments to your UI, such as hiding the action bar or
            // other navigational controls.
        }
    }
});
```

保持系统栏和UI同步是一种很好的实践方式，比如当状态栏显示或隐藏的时候进行ActionBar的显示和隐藏等等。

创建使用Material Design的应用

> 编写: allenlsy - 原文: https://developer.android.com/training/material/index.html

Material Design 是一个全面的关于视觉，动作和交互的指南，实现跨平台的设计。要在你的 Android 应用中使用 Material Design， 你需要遵从 Material Design 规格文档，来使用 Android 5.0 中新添加的组件和功能。

本课会通过以下方面教你如何创建 Material Design 设计的应用：

- Material Design 主题
- 用于卡片和列表的小组件
- 定义Shadows与Clipping视图
- 矢量 drawable
- 自定义动画

本课还将告诉你在使用 Material Design 时如何兼容 Android 5.0 (API level 21) 之前的版本。

课程

开始使用Material Design

学习如何升级应用，使用 Material Design 特性

使用 Material Design 主题

学习如何使用 Material Design 主题

用于卡片和列表的小组件

学习如何创建列表和卡片视图，使得应用和其他系统组件风格统一

定义Shadows与Clipping视图

学习如何设置 evaluation 来自定义阴影，以及创建 Clipping 视图

使用 Drawables

学习如何创建矢量 Drawable 以及如何给 drawable 资源着色

自定义动画

学习如何为视图和 Activity 切换创建自定义动画

维护兼容性

学习如何兼容 Android 5.0 以下的版本

开始使用Material Design

> 编写: allenlsy - 原文: https://developer.android.com/training/material/get-started.html

要创建一个 Material Design 应用：

1. 学习 Material Design 规格标准
2. 应用 Material Design 主题
3. 创建符合 Material Design 的 Layout 文件
4. 定义视图的 elevation 值来修改阴影
5. 使用系统组件来创建列表和卡片
6. 自定义动画

维护向下兼容性

你可以添加 Material Design 特性，同时保持对 Android 5.0 之前版本的兼容。更多信息，请参见维护兼容性章节。

使用 Material Design 更新现有应用

要更新现有应用，使其使用 Material Design，你需要翻新你的 layout 文件来遵从 Material Design 标准，并确保其包含了正确的元素高度，触摸反馈和动画。

使用 Material Design 创建新的应用

如果你要创建使用 Material Design 的新的应用，Material Design 指南提供了一套跨平台统一的设计。请遵从指南，使用新功能来进行 Android 应用的设计和开发。

应用 Material 主题

要在应用中使用 Material 主题，需要定义一个继承于 `android:Theme.Material` 的 style 文件：

```
<!-- res/values/styles.xml -->
<resources>
  <!-- your theme inherits from the material theme -->
  <style name="AppTheme" parent="android:Theme.Material">
    <!-- theme customizations -->
  </style>
</resources>
```

Material 主题提供了更新后的系统组件，使你可以设置调色板和在触摸和 Activity 切换时使用默认的动画。更多信息，请参见 Material 主题 章节。

设计你的 Layouts

另外，要应用自定义的 Material 主题，你的 layout 应该要符合 Material 设计规范。在设计 Layout 时，尤其要注意一下方面：

- 基准线网格
- Keyline
- 间隙
- 触摸目标的大小
- Layout 结构

定义视图的 Elevation

视图可以投射阴影， elevation 值决定了阴影的大小和绘制顺序。要设定 elevation 值，请使用 `android:elevation` 属性：

```
<TextView
    android:id="@+id/my_textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/next"
    android:background="@color/white"
    android:elevation="5dp" />
```

新的 `translationZ` 属性使得你可以设计临时变更 elevation 的动画。elevation 变化在做触摸反馈时很有用。

更多信息，请参见定义阴影和 Clipping 视图章节。

创建列表和卡片

RecyclerView 是一个植入性更强的 ListView，它支持不同的 layout 类型，并可以提升性能。CardView 使得你可以在卡片内显示一部分内容，并且和其他应用保持外观一致。以下是一段样例代码展示如何在 layout 中添加 CardView

```
<android.support.v7.widget.CardView
    android:id="@+id/card_view"
    android:layout_width="200dp"
    android:layout_height="200dp"
    card_view:cardCornerRadius="3dp">
    ...
</android.support.v7.widget.CardView>
```

更多信息，请参见列表和卡片章节。

自定义动画

Android 5.0 (API level 21) 包含了新的创建自定义动画 API。比如，你可以在 activity 中定义进入和退出 activity 时的动画。

```
public class MyActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // enable transitions
        getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);
        setContentView(R.layout.activity_my);
    }

    public void onSomeButtonClicked(View view) {
        getWindow().setExitTransition(new Explode());
        Intent intent = new Intent(this, MyOtherActivity.class);
        startActivity(intent,
                      ActivityOptions
                          .makeSceneTransitionAnimation(this).toBundle());
    }
}
```

当你从当前 activity 进入另一个 activity 时，退出切换动画会被调用。

想学习更多新的动画 API，参见自定义动画章节)。

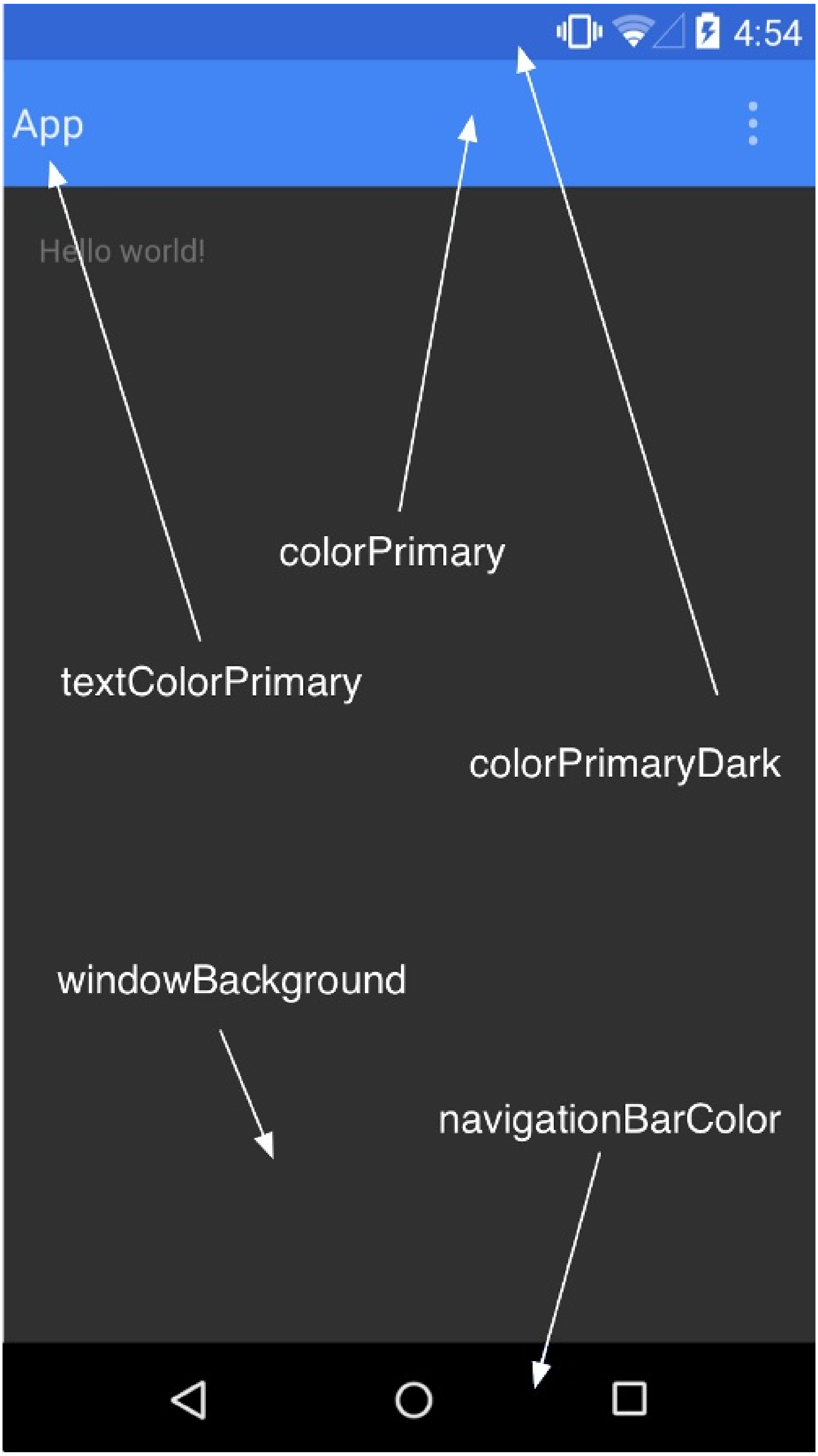
使用Material的主题

> 编写: allenlsy - 原文: https://developer.android.com/training/material/theme.html

新的 Material 主题提供：

- 系统组件，用于设定调色板
- 系统组件的触摸反馈动画
- Activity 切换动画

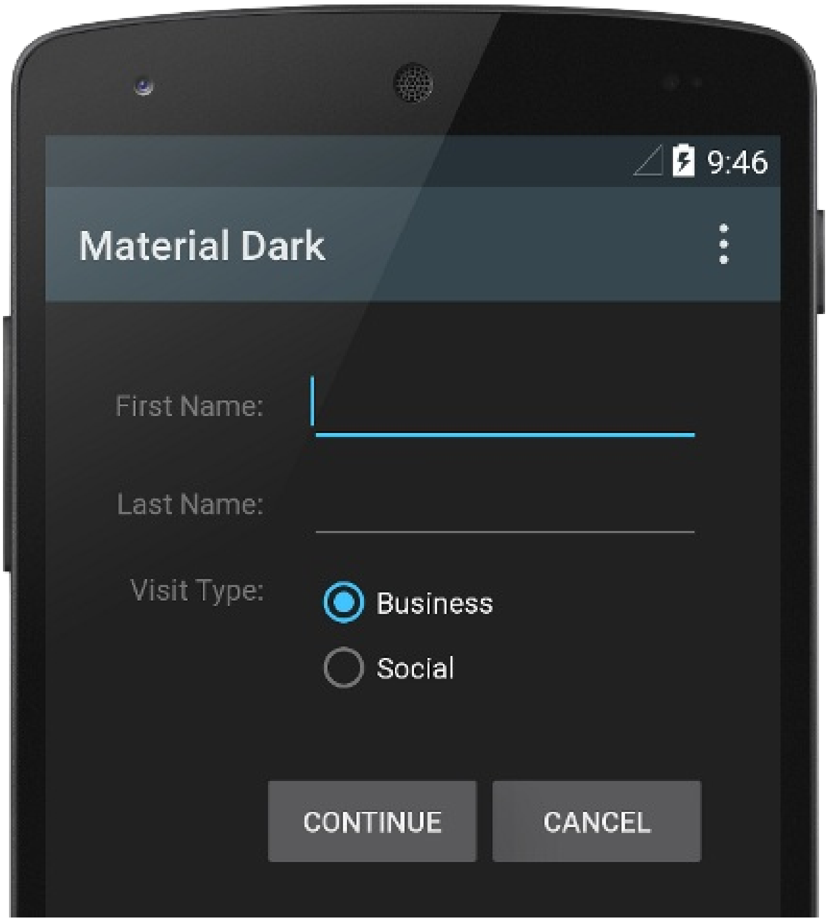
你可以根据你的品牌特征修改调色板，从而自定义 Material 主题。你可以通过主题属性调整 action bar 和状态栏的颜色，就像下图一样：

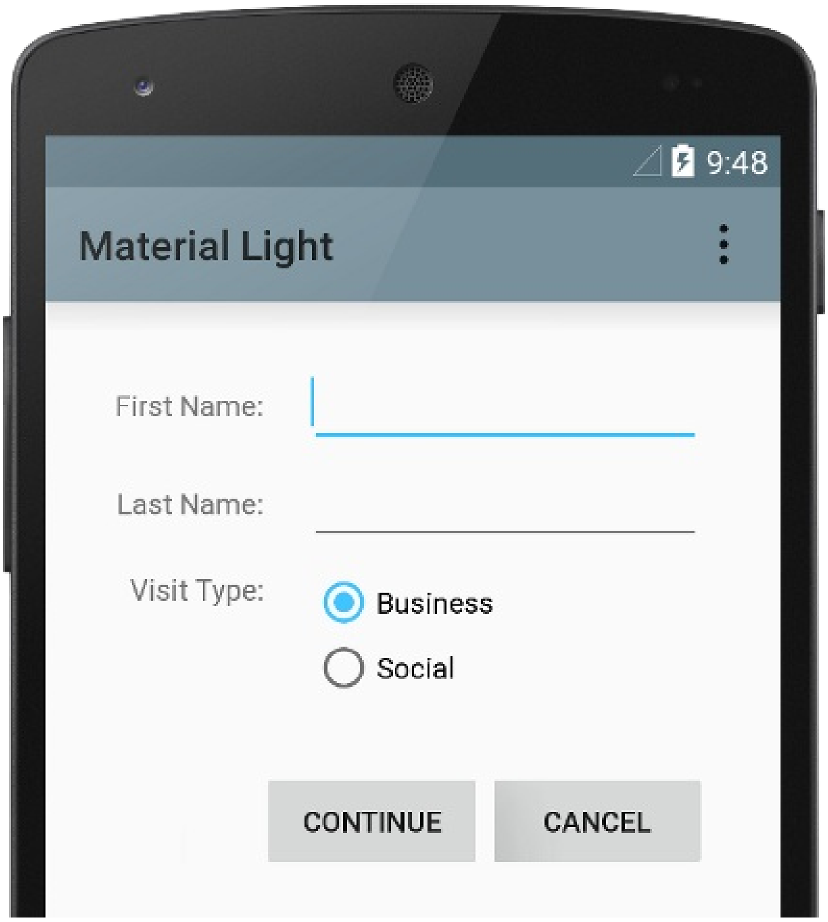


系统组件拥有新的设计和触摸反馈动画。你可以自定义调色板，反馈动画和 Activity 切换动画。

Material 主题被定义在：

- `@android:style/Theme.Material` (暗色版本)
- `@android:style/Theme.Material.Light` (亮色版本)
- `@android:style/Theme.Material.Light.DarkActionBar`





想知道可用的 Material style 的列表，可以在 API 文档中参见 R.style.

> **Note:** Material 主题只支持 Android 5.0 (API level 21) 及以上版本。v7 Support 库提供了一些组件的 Material Deisgn 样式，也支持自定义调色板。更多信息，请参见维护兼容性章节。

自定义调色板

在根据自己的品牌自定义调色板时，你需要在继承 material 主题时定义 theme 属性。

```
<resources>
  <!-- inherit from the material theme -->
  <style name="AppTheme" parent="android:Theme.Material">
    <!-- Main theme colors -->
    <!--   your app branding color for the app bar -->
    <item name="android:colorPrimary">@color/primary</item>
    <!--   darker variant for the status bar and contextual app bars -->
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <!--   theme UI controls like checkboxes and text fields -->
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

自定义状态栏

Material 主题使得你很容易自定义状态栏，你可以设定适合自己品牌的颜色，并提供足够的对比度，以显示白色的状态图标。设置状态栏颜色时，要在继承 Material 主题时设定 `android:statsBarColor` 属性。默认情况下，`android:statusBarColor` 会继承 `android:colorPrimaryDark` 的值。

你也可以在状态栏的背景上绘画。比如，你想让位于照片之上的状态栏透明，并保留一点深色渐变以确保白色图标可见。这样的话，设定 `android:statusBarColor` 属性为 `@android:color/transparent` 并调整窗口的 Flag 标记。你也可以用 `Window.setStatusBarColor()` 来实现动画或淡入淡出。

> **Note:** 状态栏必须随时保持和 primary toolbar (即顶部Actionbar，译者注) 的界线清晰。除了一种情况，即在状态栏后面显示图片或媒体内容时之外，你都要用渐变色来确保前台图标仍然可见。

当你自定义导航栏和状态栏时，要么两者都透明，要么只修改状态栏。其他情况下，导航栏应该保持黑色。

主题单独视图

XML layout 中的元素可以定义 `android:theme` 属性， 用于引用主题资源。这个属性修改了自己和子元素的主题，对于要修改局部颜色主题的情况十分有用。

创建Lists与Cards

> 编写: allenlsy - 原文: https://developer.android.com/training/material/lists-cards.html

要在应用中创建复杂的列表和使用 Material Design 的卡片列表，你可以使用 `RecyclerView` 和 `CardView` 。

创建列表

`RecyclerView` 组件是一个更高级和伸缩性更强的 ListView。这个组件是一个显示大量数据的容器，通过维护有限量的View，来达到滚动时的高效。当你的数据集在运行过程中会根据用户行为或网络事件更新时，应该使用 `RecyclerView` 。

`RecyclerView` 通过以下方式简化显示流程，并操作大量数据：

- 使用 Layout manager 来定位元素
- 为常用操作定义默认动画，比如添加或移除元素

你也可以为 RecyclerView 自定义 Layout manager 和动画。

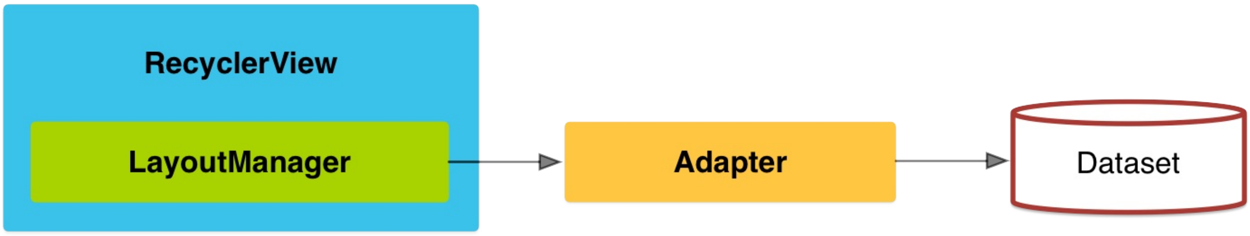
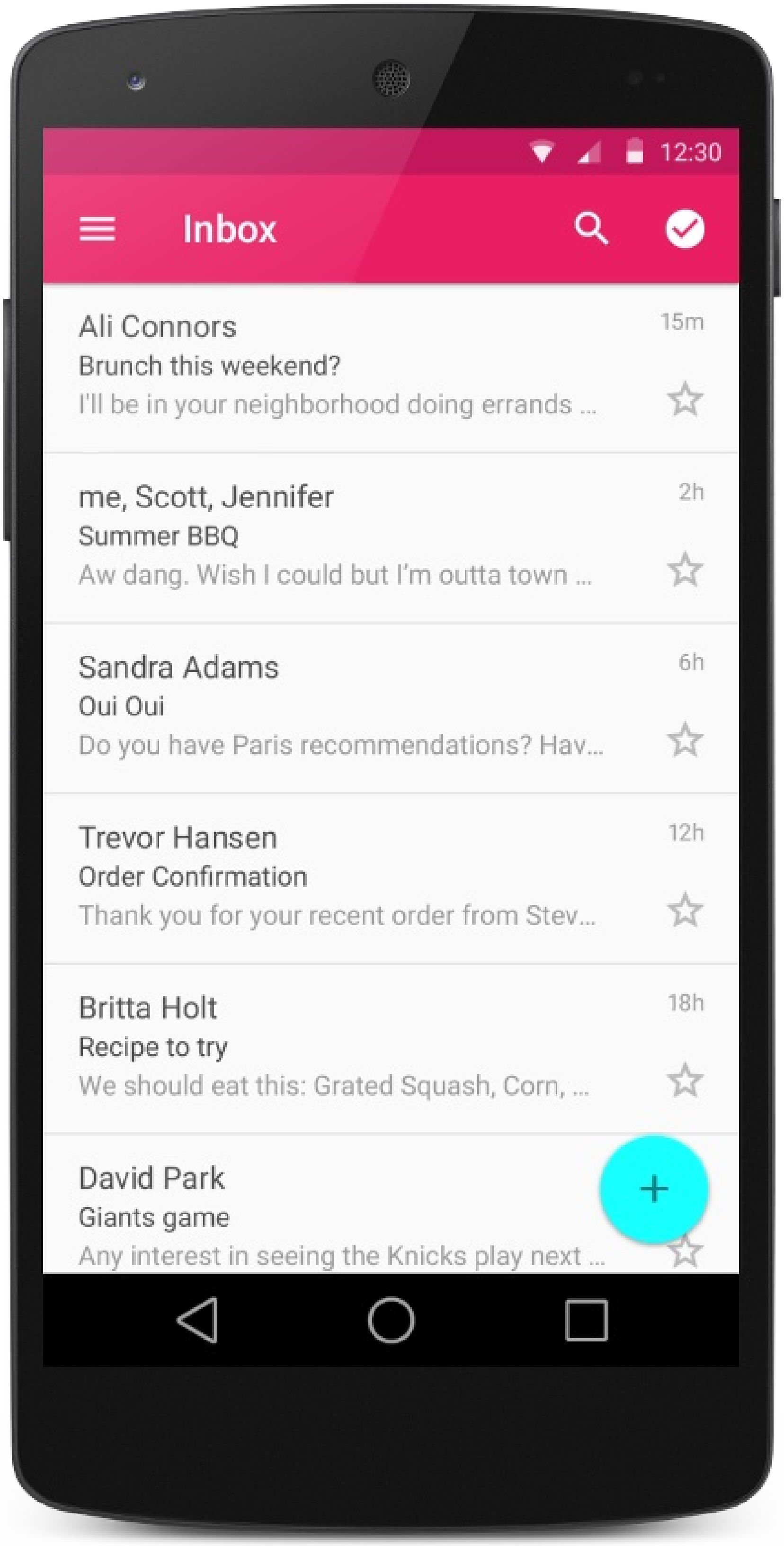


图**1**. The `RecyclerView` widget.

要使用 RecyclerView 组件，你需要定义一个 adapter 和 layout manager。创建 adapter，要继承 `RecyclerView.Adapter` 类。实现类的细节取决于你的数据集和视图类型。更多信息，请看以下样例。



Layout manager把元素视图放在 `RecyclerView` ，并决定什么时候重用不可见的元素视图。要重用（或回收）视图时，layout manager 会让 adapter 用另外的元素内容替换视图内的内容。回收 View 这个方法能提高性能，因为它避免了创建不必要的view对象，或执行昂贵的 `findViewById()` 查找。

`RecyclerView` 提供以下内建的 layout manager:

- `LinearLayoutManager` 用于显示横向或纵向的滚动列表
- `GridLayoutManager` 用于显示方格元素
- `StaggeredGridLayoutManager` 在 staggered 方格中显示元素

创建一个自定义的 layout manager，要继承于 `RecyclerView.LayoutManager` 类

动画

添加和删除元素的动画在 RecyclerView 中是默认被启用的。要自定义动画，你需要继承 `RecyclerView.ItemAnimator` 类，使用 `RecyclerView.setItemAnimator()` 方法。

例子

以下代码示例了如何添加 RecyclerView 到一个 Layout：

```
<!-- A RecyclerView with some commonly used attributes -->
<android.support.v7.widget.RecyclerView
    android:id="@+id/my_recycler_view"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

添加 RecyclerView 组件到Layout之后，获得一个到 RecyclerView 的对象，连接它到 Layout manager，再附上 adapter 用于数据显示：

```
public class MyActivity extends Activity {
    private RecyclerView mRecyclerView;
    private RecyclerView.Adapter mAdapter;
    private RecyclerView.LayoutManager mLayoutManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_activity);
        mRecyclerView = (RecyclerView) findViewById(R.id.my_recycler_view);

        // use this setting to improve performance if you know that changes
        // in content do not change the layout size of the RecyclerView
        mRecyclerView.setHasFixedSize(true);

        // use a linear layout manager
        mLayoutManager = new LinearLayoutManager(this);
        mRecyclerView.setLayoutManager(mLayoutManager);

        // specify an adapter (see also next example)
        mAdapter = new MyAdapter(myDataset);
        mRecyclerView.setAdapter(mAdapter);
    }
    ...
}
```

Adapter 支持获取数据集元素，创建元素的视图，并可以将新元素的内容去替代不可见元素视图中的内容。以下代码展示了一个简单的实现，其中的数据集包含了一个字符串数组，数据元素用 TextView 显示：

```java
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private String[] mDataset;

    // Provide a reference to the views for each data item
    // Complex data items may need more than one view per item, and
    // you provide access to all the views for a data item in a view holder
    public static class ViewHolder extends RecyclerView.ViewHolder {
        // each data item is just a string in this case
        public TextView mTextView;
        public ViewHolder(TextView v) {
            super(v);
            mTextView = v;
        }
    }

    // Provide a suitable constructor (depends on the kind of dataset)
    public MyAdapter(String[] myDataset) {
        mDataset = myDataset;
    }

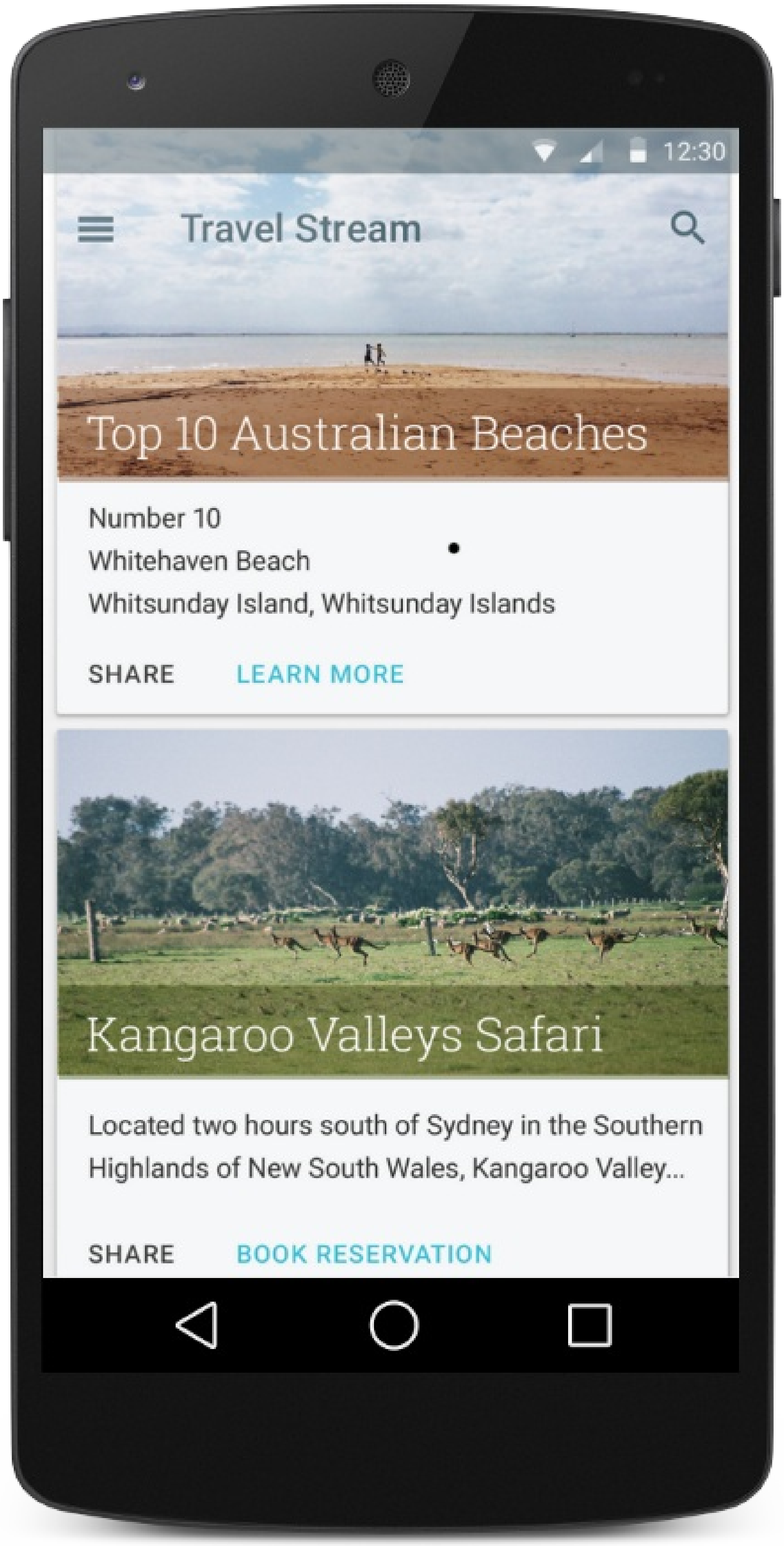
    // Create new views (invoked by the layout manager)
    @Override
    public MyAdapter.ViewHolder onCreateViewHolder(ViewGroup parent,
                                                   int viewType) {
        // create a new view
        View v = LayoutInflater.from(parent.getContext())
                               .inflate(R.layout.my_text_view, parent, false);
        // set the view's size, margins, paddings and layout parameters
        ...
        ViewHolder vh = new ViewHolder(v);
        return vh;
    }

    // Replace the contents of a view (invoked by the layout manager)
    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        // - get element from your dataset at this position
        // - replace the contents of the view with that element
        holder.mTextView.setText(mDataset[position]);

    }

    // Return the size of your dataset (invoked by the layout manager)
    @Override
    public int getItemCount() {
        return mDataset.length;
    }
}
```

创建卡片



CardView 继承于 FrameLayout 类，它可以在卡片中显示信息，并保持在不同平台上拥有统一的风格。CardView 组件可以设定阴影和圆角。

要创建一个带阴影的卡片，使用 `card_view:cardElevation` 属性。CardView 使用了Android 5.0 (API level 21)中的真实高度值以及动态阴影效果，在 5.0 以下的版本中有编程实现阴影的备选方案。更多内容，请参见保持兼容性章节。

使用以下属性来自定义CardView：

- 使用 `card_view:cardCornerRadius` 在layout中设置圆角
- 使用 `CardView.setRadius` 在代码中设置圆角
- 使用 `card_view:carBackgroundColor` 来设置背景颜色

以下代码展示如何在layout中添加CardView:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    ... >
    <!-- A CardView that contains a TextView -->
    <android.support.v7.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        android:id="@+id/card_view"
        android:layout_gravity="center"
        android:layout_width="200dp"
        android:layout_height="200dp"
        card_view:cardCornerRadius="4dp">

        <TextView
            android:id="@+id/info_text"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </android.support.v7.widget.CardView>
</LinearLayout>
```

更多信息，参见CardView的API文档。

添加依赖

RecyclerView和CardView都是v7 support 库的一部分。要使用这两个组件，在你的Gradle依赖中添加两个模块：

```
dependencies {
    ...
    compile 'com.android.support:cardview-v7:21.0.+'
    compile 'com.android.support:recyclerview-v7:21.0.+'
}
```

定义Shadows与Clipping视图

> 编写: allenlsy - 原文: https://developer.android.com/training/material/shadows-clipping.html

Material Design 引入了UI元素深度的概念。深度可以帮助用户理解每个元素的不同重要性，让用户集中注意力做手头的工作。

视图的elevation，用 Z 属性来表示，它决定了阴影的大小：更大的 Z 值可以投射出更大更柔软的阴影。Z 值较大的视图会遮盖住Z值较小的视图。不过，Z值大小不会影响视图的大小。

阴影是由被投射视图的上级视图来完成绘制，因此他受上级视图影响，附着在上级视图上。

Elevation对于创建临时上升这种动画同样很有用。

更多信息，请参见3D空间中的对象。

给视图赋Elevation值

视图的 Z 值有两个组成部分：

- elevation: 静态组成部分
- translation: 动态部分，用于动画

Z = elevation + translationZ



图**1** - 不同深度view的阴影.

在layout中设置视图的elevation，要使用 `android:elevation` 属性。要在Activity代码中设置elevation，使用 `View.setElevation()` 方法。

要设置视图的translation，使用 `View.setTranslationZ()` 方法。

新的 `ViewPropertyAnimator.z()` 和 `ViewPropertyAnimator.translationZ()` 方法使你可以很容易的实现elevation动画。更多信息，请查看ViewPropertyAnimator和属性动画开发指南。

你也可以使用 StateListAnimator 来声明动画。这非常适用于要通过状态改变来触发动画的情况，比如当用户按下按钮。更多信息，请查看Animate View State Changes（当视图状态变化的动画，译者注）。

Z值的计算单位是dp。

自定义视图的阴影和轮廓

视图背景的边界决定了阴影的形状。轮廓是一个图形对象的外围形状，决定了触摸反馈动画的ripple区域。

假设以下是个视图：

```
<TextView
    android:id="@+id/myview"
    ...
    android:elevation="2dp"
    android:background="@drawable/myrect" />
```

背景drawable定义为一个圆角的矩形：

```
<!-- res/drawable/myrect.xml -->
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">
    <solid android:color="#42000000" />
    <corners android:radius="5dp" />
</shape>
```

这个视图会投影出圆角，因为背景drawble可以决定视图轮廓。如果提供一个自定义的轮廓，会覆盖这个默认的阴影形状。

以下方式可以自定义视图的轮廓：

1. 继承 `ViewOutlineProvider` 类
2. 覆写 `getOutline()` 函数.
3. 用 `View.setOutlineProvider()` 方法来设定视图的轮廓提供者.

使用 `Outline` 类的函数，你可以创建椭圆和带圆角的矩形轮廓。视图的轮廓提供者会从视图的背景中获取轮廓。如果不想让视图投射阴影，你可以设置轮廓提供者为 null。

Clipping 视图

Clipping 视图（附着视图，译者注）使你轻松的改变视图的形状。你可以为了一致性而附着视图，也可以是为了当用户输入信息时，改变视图的形状。你可以通过 `View.setClipToOutline()` 将视图附着给一个轮廓，或使用 `android:clipToOutline` 属性。只有矩形、圆形和圆角矩形轮廓支持附着功能，你可以通过 `Outlin.canClip()` 方法来检查是否支持附着。

把视图附着给drawable的形状，要将这个drawable设置为视图的背景，并调用 `View.setClipToOutline()` 方法。

附着视图是一个昂贵的操作，所以不要对附着过的形状是进行动画。要实现这个效果，使用 Reveal Effect 动画

使用Drawables

> 编写: allenlsy - 原文: https://developer.android.com/training/material/drawables.html

使用Drawable

以下这些drawable的功能，能帮助你在应用中实现Material Design：

- Drawable染色
- 提取主色调
- 矢量Drawable

本课教你如何在应用中使用这些特性：

给 Drawable 资源染色

使用 Android 5.0 (API level 21)以上版本，你可以使用alpha mask（透明度图层，译者注）给位图和nine patches图片染色。你可以用颜色Resource或者主题属性来获取颜色（比如，`?android:attr/colorPrimary`）。通常，你只需要创建一次这些颜色asset，便可以在主题中自动匹配这些颜色。

你可以用`setTint()`方法将一种染色方式应用到`BitmapDrawable`或者`NinePatchDrawable`对象。你也在layout中使用`android:tint`和`android:initMode`属性设置染色的颜色和模式。

从图片中提取主色调

Android Support Library v21及更高版本带有`Palatte`类，可以让你从图片中提取主色调。这个类可以提取以下颜色：

- Vibrant: 亮色
- Vibrant dark: 深亮色
- Vibrant light: 浅亮色
- Muted: 暗色
- Muted dark: 深暗色
- Muted light: 浅暗色

提取这些颜色时，在你载入图片的后台线程中传入一个Bitmap对象给 `Palette.generate()` 静态方法。如果你不能使用那个线程，可以调用 `Palatte.generateAsync()` 方法，并提供一个listener。

你可以用Palette类的一个getter方法从图片获取主色调，比如 `Palette.getVibrantColor()` 。

要使用 `Palette` 类，在你的应用模块的Gradle依赖中添加以下代码：

```
dependencies {
    ...
    compile 'com.android.support:palette-v7:21.0.+'
}
```

更多信息，请参见Palette类的API文档。

创建矢量Drawable

在Android 5.0 (API level 21)以上版本中，你可以定义矢量drawable，用于无损的拉伸图片。相对于一张普通图片需要为每个不同屏幕密度的设备提供一个图片来说，一个矢量图片只需要一个asset文件。要创建矢量图片，你可以在 `<vector>` XML元素中定义形状。

以下代码定义了一个心形：

```
<!-- res/drawable/heart.xml -->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    <!-- intrinsic size of the drawable -->
    android:height="256dp"
    android:width="256dp"
    <!-- size of the virtual canvas -->
    android:viewportWidth="32"
    android:viewportHeight="32">

  <!-- draw a path -->
  <path android:fillColor="#8fff"
      android:pathData="M20.5,9.5
                        c-1.955,0,-3.83,1.268,-4.5,3
                        c-0.67,-1.732,-2.547,-3,-4.5,-3
                        C8.957,9.5,7,11.432,7,14
                        c0,3.53,3.793,6.257,9,11.5
                        c5.207,-5.242,9,-7.97,9,-11.5
                        C25,11.432,23.043,9.5,20.5,9.5z" />
</vector>
```

矢量图片在Android中用VectorDrawable对象来表示。更多关于 `pathData` 语法的信息，请看SVG Path的文档。更多关于矢量drawable动画的信息，请参见矢量drawable动画。

自定义动画

> 编写: allenlsy - 原文: https://developer.android.com/training/material/animations.html

Material Design中的动画对用户的动作进行反馈，并提供在整个交互过程中的视觉连续性。Material 主题为按钮和Activity切换提供一些默认的动画，Android 5.0 (API level 21) 及以上版本支持自定义这些动画并创建新动画：

- 触摸反馈
- 圆形填充
- Activity 切换动画
- 曲线形动作
- 视图状态变换

自定义触摸反馈

Material Design中的触摸反馈，是在用户与UI元素交互时，提供视觉上的即时确认。按钮的默认触摸反馈动画使用了新的 `RippleDrawable` 类，它在按钮状态变换时产生波纹效果。

大多数情况下，你需要在你的 XML 文件中设定视图的背景来实现这个功能：

- `?android:attr/selectableItemBackground` 用于有界Ripple动画
- `?android:attr/selectableItemBackgroundBorderless` 用于越出视图边界的动画。它会被绘制在最近的且不是全屏的父视图上。

> **Note**：`selectableItemBackgroundBorderless` 是 API level 21 新加入的属性

另外，你可以使用 `ripple` 元素在XML资源文件中定义一个 `RippleDrawable` 。

你可以给 `RippleDrawable` 赋予一个颜色。要改变默认的触摸反馈颜色，使用主题的 `android:colorControlHighlight` 属性。

更多信息，参见 `RippleDrawable` 类的API文档。

使用填充效果（Reveal Effect）

填充效果在UI元素出现或隐藏时，为用户提供视觉连续性。`ViewAnimationUtils.createCircularReveal()` 方法可以使用一个附着在视图上的圆形，显示或隐藏这个视图。

要用此效果显示一个原本不可见的视图：

```
// previously invisible view
View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = (myView.getLeft() + myView.getRight()) / 2;
int cy = (myView.getTop() + myView.getBottom()) / 2;

// get the final radius for the clipping circle
int finalRadius = myView.getWidth();

// create and start the animator for this view
// (the start radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, 0, finalRadius);
anim.start();
```

要用此效果隐藏一个原本可见的视图：

```
// previously visible view
final View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = (myView.getLeft() + myView.getRight()) / 2;
int cy = (myView.getTop() + myView.getBottom()) / 2;

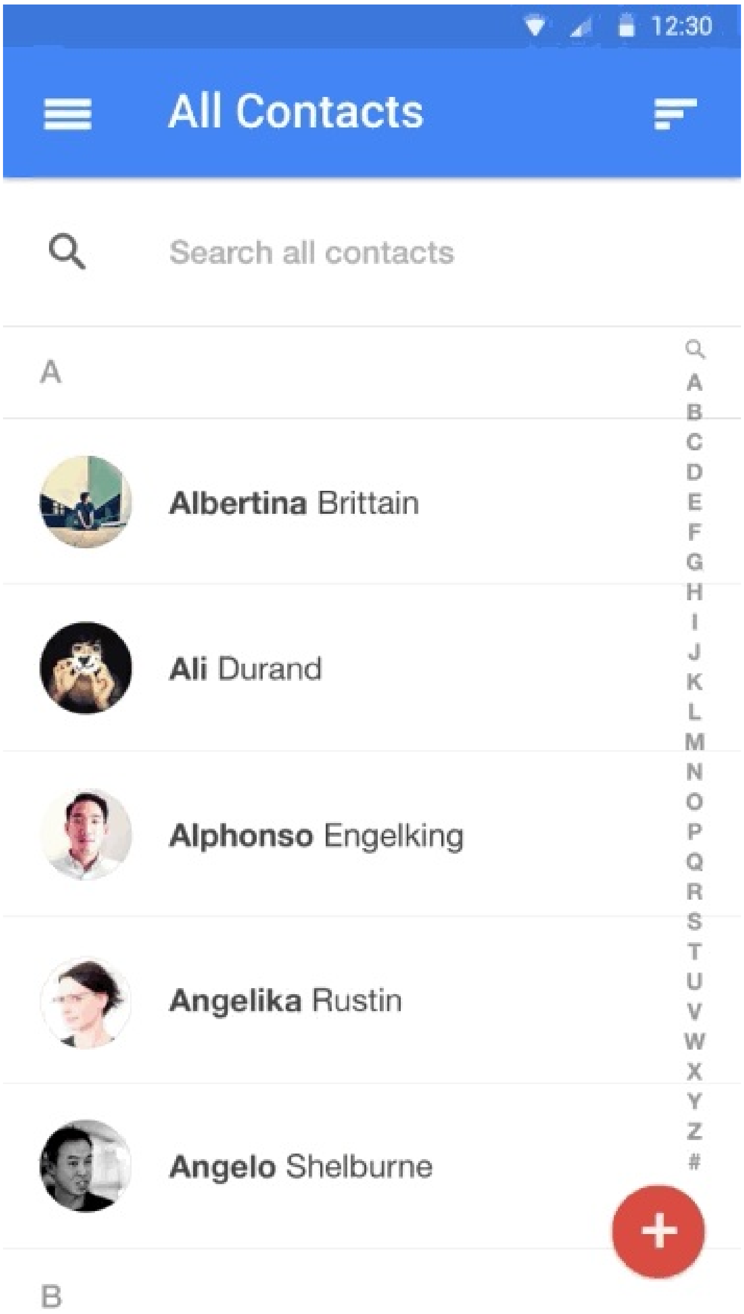
// get the initial radius for the clipping circle
int initialRadius = myView.getWidth();

// create the animation (the final radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, initialRadius, 0);

// make the view invisible when the animation is done
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
        myView.setVisibility(View.INVISIBLE);
    }
});

// start the animation
anim.start();
```

自定义Activity切换效果



Material Design中的Activity切换，当不同Activity之间拥有共有元素，则可以通过不同状态之间的动画和形变提供视觉上的连续性。你可以为共有元素设定进入和退出Activity时的自定义动画。

- 入场变换决定视图如何入场。比如，在爆炸式入场变换中，视图从场外飞到屏幕中央。
- 出场变换决定视图如何退出。比如，在爆炸式出场变换中，视图从屏幕中央飞出场外。
- 共有元素的变换决定一个共有视图在两个Activity之间如何变换。比如，如果两个activity有同一张图片，但是放在不同位置，以及拥有不同大小，变更图片 变换会流畅的把图片移到相应位置，同时缩放图片大小。

Android 5.0 (API level 21) 支持这些入场和退出变换：

- 爆炸 - 把视图移入或移出场景的中间
- 滑动 - 把视图从场景边缘移入或移出

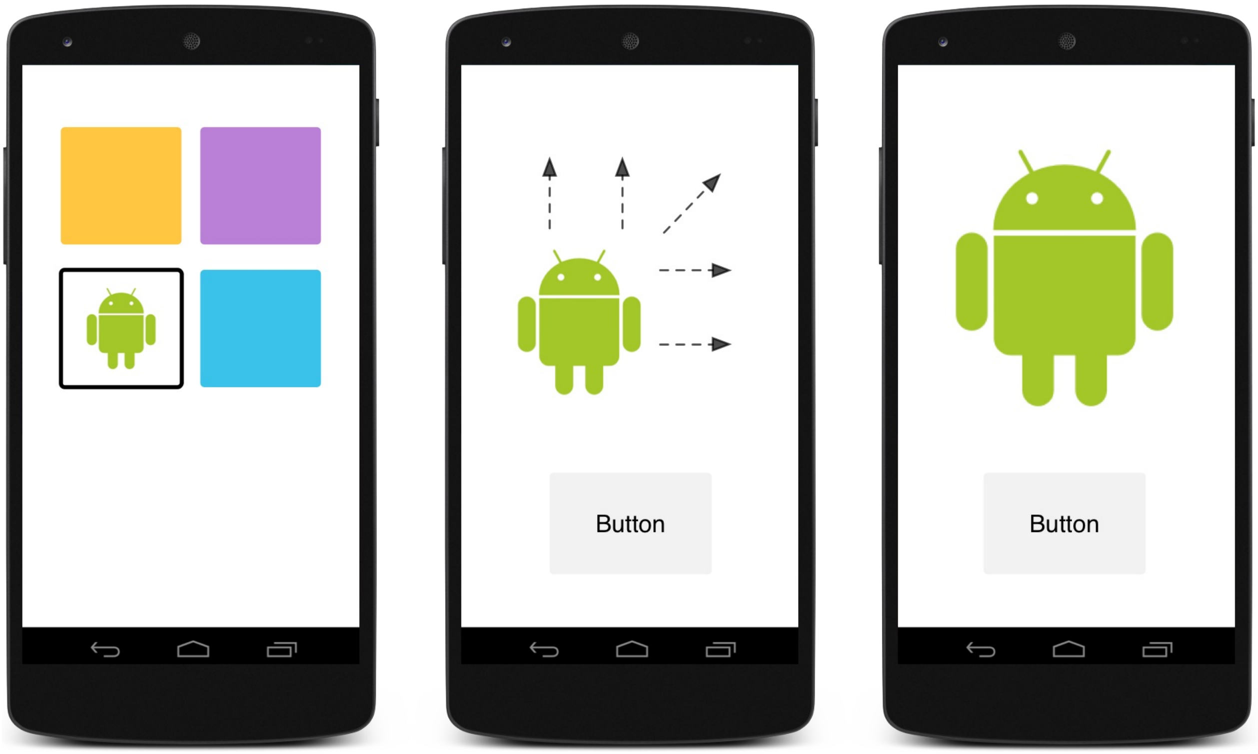
- 淡入淡出 - 通过改变透明度添加或移除元素

任何继承于 `Visibility` 类的变换，都支持被用于入场或退出变换。更多信息，请参见 `Transition` 类的API文档。

Android 5.0 (API level 21) 还支持这些共有元素变换效果：

- **changeBounds** - 对目标视图的外边界进行动画
- **chagneClipBounds** - 对目标视图的附着物的外边界进行动画
- **changeTransform** - 对目标视图进行缩放和旋转
- **changeImageTransform** - 对目标图片进行缩放

当你在应用中进行activity 变换时，默认的淡入淡出效果会被用在进入和退出activity的过程中。



Activity 1 Scene Transition Animation (common element) **Activity 2**

自定义切换

首先，当你继承Material主题的style时，要通过 `android:windowContentTransitions` 属性来开启窗口内容变换功能。你也可以在style定义中声明进入、退出和共有元素切换：

```
<style name="BaseAppTheme" parent="android:Theme.Material">
  <!-- enable window content transitions -->
  <item name="android:windowContentTransitions">true</item>

  <!-- specify enter and exit transitions -->
  <item name="android:windowEnterTransition">@transition/explode</item>
  <item name="android:windowExitTransition">@transition/explode</item>

  <!-- specify shared element transitions -->
  <item name="android:windowSharedElementEnterTransition">
    @transition/change_image_transform</item>
  <item name="android:windowSharedElementExitTransition">
    @transition/change_image_transform</item>
</style>
```

例子中的 `change_image_transform` 切换定义如下：

```
<!-- res/transition/change_image_transform.xml -->
<!-- (see also Shared Transitions below) -->
<transitionSet xmlns:android="http://schemas.android.com/apk/res/android">
  <changeImageTransform/>
</transitionSet>
```

`changeImageTransform` 元素对应 `ChangeImageTransform` 类。更多信息，请参见 `Transition` 类的API文档。

要在代码中启用窗口内容切换，调用 `Window.requestFeature()` 函数：

```
// inside your activity (if you did not enable transitions in your theme)
getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);

// set an exit transition
getWindow().setExitTransition(new Explode());
```

要声明变换类型，就要在 `Transition` 对象上调用以下函数：

- `Window.setEnterTransition()`
- `Window.setExitTransition()`
- `Window.setSharedElementEnterTransition()`
- `Window.setSharedElementExitTransition()`

`setExitTransition()` 和 `setSharedElementExitTransition()` 函数为activity定义了退出变换效果。`setEnterTransition()` 和 `setSharedElementEnterTransition()` 函数定义了进入activity的变换效果。

要获得切换的全部效果，你必须在出入的两个activity中都开启窗口内容切换。否则，调用的activity会使用退出效果，但是接着你会看到一个传统的窗口切换（比如缩放或淡入淡出）。

要尽早开始入场切换，可以在被调用的Activity上使用 `Window.setAllowEnterTransitionOverlap()` 。它可以使你拥有更戏剧性的入场切换。

使用切换启动一个Activity

如果你开启Activity入场和退出效果，那么当你在用如下方法开始Activity时，切换效果会被应用：

```
startActivity(intent,
              ActivityOptions.makeSceneTransitionAnimation(this).toBundle());
```

如果你为第二个Activity设定了入场变换，变换也会在activity开始时被启用。要在开始另一个acitivity时禁用变换，可以给bundle的选项提供一个 `null` 对象：

启动一个拥有共用元素的Activity

要在两个拥有共用元素的activity间进行切换动画：

1. 在主题中开启窗口内容切换
2. 在style中定义共有元素切换
3. 将切换定义为一个XML 资源文件
4. 使用 `android:transitionName` 属性在两个layout文件中给共有元素赋予同一个名字
5. 使用 `ActivityOptions.makeSceneTransitionAnimation()` 方法

```
// get the element that receives the click event
final View imgContainerView = findViewById(R.id.img_container);

// get the common element for the transition in this activity
final View androidRobotView = findViewById(R.id.image_small);

// define a click listener
imgContainerView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(this, Activity2.class);
        // create the transition animation - the images in the layouts
        // of both activities are defined with android:transitionName="robot"
        ActivityOptions options = ActivityOptions
            .makeSceneTransitionAnimation(this, androidRobotView, "robot");
        // start the new activity
        startActivity(intent, options.toBundle());
    }
});
```

对于用代码编写的共有动态视图，使用 `View.setTransitionName()` 方法来在两个activity中定义共有元素。

要在第二个activity结束时进行逆向的场景切换动画，调用 `Activity.finishAfterTransition()` 方法，而不是 `Activity.finish()` 。

开始一个拥有多个共有元素的Activity

要在拥有多个共有元素的activity之间使用变换动画，就要用 `android:transitionName` 属性在两个layout中定义这个共有元素（或在两个Activity中使用 `View.setTransitionName()` 方法），再创建 `ActivityOptions` 对象：

```
ActivityOptions options = ActivityOptions.makeSceneTransitionAnimation(this,
        Pair.create(view1, "agreedName1"),
        Pair.create(view2, "agreedName2"));
```

使用曲线动画

Material Design中的动画可以表示为基于时间插值和空间移动模式的曲线。在Android 5.0 (API level 21)以上版本中，你可以为动画定义时间曲线和曲线动画模式。

`PathInterpolator` 类是一个基于贝泽尔曲线或 `Path` 对象的新的插值方法。插值方法 是一个定义在 1x1 正方形中的曲线函数图像，其始末两点分别在(0,0)和（1,1)，一个用构造函数定义的控制点。你也可以使用XML资源文件定义一个插值方法：

```
<pathInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:controlX1="0.4"
    android:controlY1="0"
    android:controlX2="1"
    android:controlY2="1"/>
```

Material Design标准中，系统提供了三种基本的曲线：

- `@interpolator/fast_out_linear_in.xml`
- `@interpolator/fast_out_slow_in.xml`
- `@interpolator/linear_out_slow_in.xml`

你可以将一个 `PathInterpolator` 对象传给 `Animator.setInterpolator()` 方法。

`ObjectAnimator` 类有一个新的构造函数，使你可以沿一条路径使用多个属性来在坐标系中进行变换。比如，以下animator（动画器，译者注）使用一个 `Path` 对象来改变一个试图的X和Y属性：

```
ObjectAnimator mAnimator;
mAnimator = ObjectAnimator.ofFloat(view, View.X, View.Y, path);
...
mAnimator.start();
```

基于视图状态改变的动画

`StateListAnimator` 类是你可以定义在视图状态改变启动的Animator（动画器，译者注）。以下例子展示如何在XML文件中定义 `StateListAnimator` ：

```
<!-- animate the translationZ property of a view when pressed -->
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_pressed="true">
    <set>
      <objectAnimator android:propertyName="translationZ"
        android:duration="@android:integer/config_shortAnimTime"
        android:valueTo="2dp"
        android:valueType="floatType"/>
        <!-- you could have other objectAnimator elements
             here for "x" and "y", or other properties -->
    </set>
  </item>
  <item android:state_enabled="true"
    android:state_pressed="false"
    android:state_focused="true">
    <set>
      <objectAnimator android:propertyName="translationZ"
        android:duration="100"
        android:valueTo="0"
        android:valueType="floatType"/>
    </set>
  </item>
</selector>
```

要把视图改变Animator关联到一个视图，就要在XML资源文件的selector元素上定义一个Animator，并把此Animator赋值给视图的 `android:stateListAnimator` 属性。要想在Java代码中将状态列表Animator赋值给视图，使用 `AnimationInflater.loadStateListAnimator()` 函数，并用 `View.setStateListAnimator()` 函数把Animator赋值给你的视图。

当你的主题继承于Material Theme的时候，Button默认会有一个Z值动画。为了避免Button的Z值动画，设定它的 `android:stateListAnimator` 属性为 `@null` 。

`AnimatedStateListDrawable` 类使你可以创建一个在视图状态变化之间显示动画的drawable。有一些Android 5.0系统组件默认已经使用了这些动画。下面的例展示如何在XML资源文件中定义AnimatedStateListDrawable：

```
<!-- res/drawable/myanimstatedrawable.xml -->
<animated-selector
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- provide a different drawable for each state-->
    <item android:id="@+id/pressed" android:drawable="@drawable/drawableP"
        android:state_pressed="true"/>
    <item android:id="@+id/focused" android:drawable="@drawable/drawableF"
        android:state_focused="true"/>
    <item android:id="@id/default"
        android:drawable="@drawable/drawableD"/>

    <!-- specify a transition -->
    <transition android:fromId="@+id/default" android:toId="@+id/pressed">
        <animation-list>
            <item android:duration="15" android:drawable="@drawable/dt1"/>
            <item android:duration="15" android:drawable="@drawable/dt2"/>
            ...
        </animation-list>
    </transition>
    ...
</animated-selector>
```

动画矢量 Drawables

矢量Drawable是可以无损缩放的。 `AnimatedVectorDrawable` 类是你可以操作矢量Drawable。

你通常在3个XML文件中定义动画矢量Drawable：

- 在 `res/drawable/` 中用 `<vector>` 定义一个矢量drawable
- 在 `res/drawable/` 中用 `<animated-vector>` 定义一个动画矢量drawable
- 在`res/anim/'中定义一个或多个Animator

动画矢量drawable可以用在 `<group>` 和 `<path>` 元素的属性上。 `<group>` 元素定义了一些path或者subgroup， `<path>` 定义了一条被绘画的路径。

当你想要定义一个动画的矢量drawable时，使用 `android:name` 属性来为group和path赋值一个唯一的名字(name)，这样你可以通过animator的定义找到他们。比如：

```
<!-- res/drawable/vectordrawable.xml -->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:height="64dp"
    android:width="64dp"
    android:viewportHeight="600"
    android:viewportWidth="600">
    <group
        android:name="rotationGroup"
        android:pivotX="300.0"
        android:pivotY="300.0"
        android:rotation="45.0" >
        <path
            android:name="v"
            android:fillColor="#000000"
            android:pathData="M300,70 l 0,-70 70,70 0,0 -70,70z" />
    </group>
</vector>
```

动画矢量drawable的定义是通过name属性来找到视图组(group)和路径(path)的：

```
<!-- res/drawable/animvectordrawable.xml -->
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
  android:drawable="@drawable/vectordrawable" >
    <target
        android:name="rotationGroup"
        android:animation="@anim/rotation" />
    <target
        android:name="v"
        android:animation="@anim/path_morph" />
</animated-vector>
```

动画的定义代表 `ObjectAnimator` 或者 `AnimatorSet` 对象。例子中第一个animator将目标组旋转了360度。

```
<!-- res/anim/rotation.xml -->
<objectAnimator
    android:duration="6000"
    android:propertyName="rotation"
    android:valueFrom="0"
    android:valueTo="360" />
```

第二个animator将矢量drawable的路径从一个形状(morph)变形到另一个。两个路径都必须是可以形变的：他们必须有相同数量的命令，每个命令必须有相同数量的参数

```
<!-- res/anim/path_morph.xml -->
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <objectAnimator
        android:duration="3000"
        android:propertyName="pathData"
        android:valueFrom="M300,70 l 0,-70 70,70 0,0   -70,70z"
        android:valueTo="M300,70 l 0,-70 70,0  0,140 -70,0 z"
        android:valueType="pathType" />
</set>
```

更多信息，请参考 `AnimatedVectorDrawable`)的API指南。

维护兼容性

> 编写: allenlsy - 原文: https://developer.android.com/training/material/compatibility.html

有些Material Design特性，比如主题和自定义Activits切换效果等，只在Android 5.0 (API level 21) 以上中可用。不过，你仍然可以使用这些特性实现Material Design，并保持对旧版本Android 系统的兼容。

定义备选Style

你可以配置你的应用，在支持Material Design的设备上使用Material主题，在旧版本Android上使用旧的主题：

1. 在 `res/values/styles/xml` 中定义一个主题继承自旧主题（比如Holo）
2. 在 `res/values-v21/styles.xml` 中定义一个同名的主题，继承自Material 主题
3. 在 `AndroidManifest.xml` 中，将这个主题设置为应用的主题

> **Note:** 如果你的应用设置了一个主题，但是没有提供备选Style，你可能无法在低于Android 5.0版本的系统中运行应用。

提供备选layout

如果你根据Material Design设计的应用的Layout中没有使用任何Android 5.0 (API level 21)中新的XML属性，他们在旧版本Android中就能正常工作。否则，你要提供备选Layout。你可以在备选Layout中定义你的应用在旧版本系统中的界面。

在 `res/layout-v21/` 中定义Android 5.0 (API level 21) 以上系统的Layout，在 `res/layout` 中定义早前版本Android的Layout。比如， `res/layout/my_activity.xml` 是对于 `res/layout-v21/my_activity.xml` 的一个备选Layout。

为了避免代码重复，在 `res/values` 中定义style，然后在 `res/values-v21` 中修改新API需要的style。使用style的继承，在 `res/values/` 中定义父style，在 `res/values-v21/` 中继承。

使用 Support Library

v7 support libraries r21 及更高版本包含了以下Material Design 特性：

- 当你应用一个 `Theme.AppCompat` 主题时， 会得到为一些系统控件准备的 Material Design style

- `Theme.AppCompat` 主题包含调色板主体属性
- `RecyclerView` 组件用于显示数据集
- `CardView` 组件用于创建卡片
- `Palette` 类用于从图片提取主色调

系统组件

`Theme.AppCompat` 主题中提供了这些组件的 Material Design style：

- EditText
- Spinner
- CheckBox
- RadioButton
- SwitchCompat
- CheckedTextView

调色板

要获取Material Design style，并用v7 support library自定义调色板，就要应用以下中的一个Theme.AppCompat主题：

```
<!-- extend one of the Theme.AppCompat themes -->
<style name="Theme.MyTheme" parent="Theme.AppCompat.Light">
    <!-- customize the color palette -->
    <item name="colorPrimary">@color/material_blue_500</item>
    <item name="colorPrimaryDark">@color/material_blue_700</item>
    <item name="colorAccent">@color/material_green_A200</item>
</style>
```

列表和卡片

`RecyclerView` 和 `CardView` 组件可通过v7 support libraries支持旧版本Android，但有以下限制：

- CardView需要编程实现阴影和其他的padding
- CardView不能将附着与原件有重合部分的子视图

依赖

要在Android 5.0之前的版本使用这些特性，需要在项目的Gradle依赖中加入Android v7 support library:

```
dependencies {
    compile 'com.android.support:appcompat-v7:21.0.+'
    compile 'com.android.support:cardview-v7:21.0.+'
    compile 'com.android.support:recyclerview-v7:21.0.+'
}
```

检查系统版本

以下特性只在Android 5.0 (API level 21) 及以上版本中可用：

- Activity 切换动画
- 触摸反馈
- Reveal 动画（填充动画效果，译者注）
- 基于路径的动画
- 矢量drawable
- Drawable染色

要保持向下兼容，请在使用这些特性是，使用以下代码在运行时检查系统版本：

```
// Check if we're running on Android 5.0 or higher
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // Call some material design APIs here
} else {
    // Implement this feature without material design
}
```

> **Note:**要声明应用支持哪些Android 版本，在manifest文件中使用 `android:minSdkVersion` 和 `android:targetSdkVersion` 属性。要在Android 5.0中使用Material Design特性，设置 `android:targetSdkVersion` 属性为21。更多信息，参见 `<uses-sdk>` API指南。

用户输入

> 编写:kesenhoo - 原文:http://developer.android.com/training/best-user-input.html

本课程涵盖的主题包括多种多样用户输入，例如触摸屏幕手势、通过屏幕输入法和硬件键盘的文本输入。

使用触摸手势

介绍如何编写允许用户通过触摸手势与触摸屏幕进行交互的app程序。

处理键盘输入事件

介绍在软输入方法下（如屏幕键盘按键情况下）程序的响应表现和执行动作，以及如何优化在硬件键盘按键下的用户体验。

兼容游戏控制器

介绍如何编写支持游戏控制器的app。

使用触摸手势

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/index.html

本章节讲述，如何编写一个允许用户通过触摸手势进行交互的app。Android提供了各种各样的API，来帮助我们创建和检测手势。

尽管对于一些基本的操作来说，我们的app不应该依赖于触摸手势（因为某些情况下手势是不用的）。但为我们的app添加基于触摸的交互，将会大大地提高app的可用性和吸引力。

为了给用户提供一致的、符合直觉的使用体验，我们的app应该遵守Android触摸手势的惯常做法。手势设计指南介绍了在Android app中，如何使用常用的手势。同样，设计指南也提供了触摸反馈的相关内容。

Lessons

检测常用的手势

学习如何通过使用GestureDetector来检测基本的触摸手势，如滑动、惯性滑动以及双击。

追踪手势移动

学习如何追踪手势移动。

Scroll手势动画

学习如何使用scrollers（Scrollers以及OverScroll）来产生滚动动画，以响应触摸事件。

处理多触摸手势

学习如何检测多点(手指)触摸手势。

拖拽与缩放

学习如何实现基于触摸的拖拽与缩放。

管理**ViewGroup**中的触摸事件

学习如何在ViewGroup中管理触摸事件，以确保事件能被正确地分发到目标views上。

检测常用的手势

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/detector.html

当用户把用一根或多根手指放在触摸屏上，并且应用把这样的触摸方式解释为特定的手势时，“触摸手势”就发生了。相应地，检测手势也就有以下两个阶段：

1. 收集触摸事件的相关数据。
2. 分析这些数据，看它们是否符合app所支持的手势的标准。

Support Library 中的类

本节课程的示例程序使用了GestureDetectorCompat和MotionEventCompat类。这些类都是在Support Library 中定义的。如果有可能的情况话，我们应该使用 Support Library 中的类，来为运行着Android1.6及以上版本系统的设备提供兼容性功能。需要注意的一点是，MotionEventCompat并不是MotionEvent的替代品，而是提供了一些静态工具类函数。我们可以把MotionEvent对象作为参数传递给这些工具类函数，来获得与触摸事件相关的动作(action)。

收集数据

当用户把用一根或多根手指放在触摸屏上时，会触发 View 上用于接收触摸事件的onTouchEvent() 回调函数。对于一系列连续的、最终会被识别为一种手势的触摸事件（位置、压力、大小、添加另一根手指等等），onTouchEvent()会被调用若干次。

当用户第一次触摸屏幕时，手势就开始了。其后系统会持续地追踪用户手指的位置，在用户手指全都离开屏幕时，手势结束。在整个交互期间，被分发给 onTouchEvent() 函数的MotionEvent 对象，提供了每次交互的详细信息。我们的app可以使用 MotionEvent 提供的这些数据，来判断某种特定的手势是否发生了。

为Activity或View捕获触摸事件

为了捕获Activity或View中的触摸事件，我们可以重写onTouchEvent()回调函数。

接下来的代码段使用了getActionMasked()函数，来从 `event` 参数中抽取出用户执行的动作。它提供了一些原始的触摸数据，我们可以使用这些数据，来判断某个特定手势是否发生了。

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same approach if
// you were subclassing a View.
@Override
public boolean onTouchEvent(MotionEvent event){

        int action = MotionEventCompat.getActionMasked(event);

        switch(action) {
                case (MotionEvent.ACTION_DOWN) :
                Log.d(DEBUG_TAG,"Action was DOWN");
                return true;
        case (MotionEvent.ACTION_MOVE) :
                Log.d(DEBUG_TAG,"Action was MOVE");
                return true;
        case (MotionEvent.ACTION_UP) :
                Log.d(DEBUG_TAG,"Action was UP");
                return true;
        case (MotionEvent.ACTION_CANCEL) :
                Log.d(DEBUG_TAG,"Action was CANCEL");
                return true;
        case (MotionEvent.ACTION_OUTSIDE) :
                Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                        "of current screen element");
                return true;
        default :
                return super.onTouchEvent(event);
        }
}
```

然后，我们可以对这些事件做些自己的处理，以判断某个手势是否出现了。这种是针对自定义手势，我们所需要进行的处理。然而，如果我们的app仅仅需要一些常见的手势，如双击，长按，快速滑动（fling）等，那么我们可以使用GestureDetector类来完成。 GestureDetector可以让我们简单地检测常见手势，并且无需自行处理单个触摸事件。相关内容将会在下面的检测手势中讨论。

捕获单个view的触摸事件

作为onTouchEvent()的一种替换方式，我们也可以使用 setOnTouchListener() 函数，来把 View.OnTouchListener 关联到任意的View上。这样可以在不继承已有的 View 的情况下，也能监听触摸事件。比如:

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

创建listener对象时，注意 ACTION_DOWN 事件返回 `false` 的情况。如果返回 `false` ，会让listener对象接收不到后续的ACTION_MOVE、ACTION_UP等系列事件。这是因为ACTION_DOWN事件是所有触摸事件的开端。

如果我们正在写一个自定义View，我们也可以像上面描述的那样重写onTouchEvent()函数。

检测手势

Android提供了GestureDetector类来检测常用的手势。它所支持的手势包括onDown()、onLongPress()、onFling() 等。我们可以把GestureDetector和上面描述的onTouchEvent()函数结合在一起使用。

检测所有支持的手势

当我们实例化一个GestureDetectorCompat对象时，需要一个实现了GestureDetector.OnGestureListener接口的类作为参数。当某个特定的触摸事件发生时，GestureDetector.OnGestureListener就会通知用户。为了让我们的GestureDetector对象能到接收到触摸事件，我们需要重写 View 或 Activity 的 onTouchEvent() 函数，并且把所有捕获到的事件传递给 detector 实例。

接下来的代码段中， `on<TouchEvent>` 型的函数的返回值是 `true` ，意味着我们已经处理完这个触摸事件了。如果返回 `false` ，则会把事件沿view栈传递，直到触摸事件被成功地处理了。

运行下面的代码段，来了解当我们与触摸屏交互时，动作（action）是如何触发的，以及每个触摸事件MotionEvent中的内容。我们也会意识到，一个简单的交互会产生多少的数据。

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;

    // Called when the activity is first created.
    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Instantiate the gesture detector with the
        // application context and an implementation of
        // GestureDetector.OnGestureListener
        mDetector = new GestureDetectorCompat(this,this);
        // Set the gesture detector as the double tap
        // listener.
        mDetector.setOnDoubleTapListener(this);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        // Be sure to call the superclass implementation
        return super.onTouchEvent(event);
    }

    @Override
    public boolean onDown(MotionEvent event) {
        Log.d(DEBUG_TAG,"onDown: " + event.toString());
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
            float velocityX, float velocityY) {
        Log.d(DEBUG_TAG, "onFling: " + event1.toString()+event2.toString());
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX,
            float distanceY) {
        Log.d(DEBUG_TAG, "onScroll: " + e1.toString()+e2.toString());
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapUp: " + event.toString());
        return true;
```

```
    }

    @Override
    public boolean onDoubleTap(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTap: " + event.toString());
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTapEvent: " + event.toString());
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapConfirmed: " + event.toString());
        return true;
    }
}
```

检测部分支持的手势

如果我们只想处理几种手势，那么可以选择继承 GestureDetector.SimpleOnGestureListener 类，而不是实现 GestureDetector.OnGestureListener 接口。

GestureDetector.SimpleOnGestureListener 类实现了所有的 `on<TouchEvent>` 型函数，其中，这些函数都返回 `false` 。因此，我们可以仅仅重写我们需要的函数。比如，下面的代码段中，创建了一个继承自 GestureDetector.SimpleOnGestureListener 的类，并重写了 onFling() 和 onDown() 函数。

无论我们是否使用GestureDetector.OnGestureListener类，最好都实现 onDown() 函数并且返回 `true` 。这是因为所有的手势都是由 onDown() 消息开始的。如果让 onDown() 函数返回 `false` ，就像GestureDetector.SimpleOnGestureListener类中默认实现的那样，系统会假定我们想忽略剩余的手势，GestureDetector.OnGestureListener中的其他函数也就永远不会被调用。这可能会导致我们的app出现意想不到的问题。仅仅当我们真的想忽略全部手势时，我们才应该让 onDown() 函数返回 `false` 。

```
public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG,"onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent event2,
                float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " + event1.toString()+event2.toString());
            return true;
        }
    }
}
```

追踪手势移动

> 编写:Andrwyw - 原文：http://developer.android.com/training/gestures/movement.html

本节课程讲述如何追踪手势移动。

每当当前的触摸位置、压力、大小发生变化时，ACTION_MOVE事件都会触发onTouchEvent()函数。正如检测常用的手势中描述的那样，触摸事件全部都记录在onTouchEvent()函数的MotionEvent参数中。

因为基于手指的触摸的交互方式并不总是非常精确，所以检测触摸事件更多的是基于手势移动，而非简单地基于触摸。为了帮助app区分基于移动的手势（如滑动）和非移动手势（如简单地点击），Android引入了“touch slop”的概念。Touch slop是指，在被识别为基于移动的手势前，用户触摸可移动的那一段像素距离。关于这一主题的更多讨论，可以在管理ViewGroup中的触摸事件中查看。

根据应用的需求，有多种追踪手势移动的方式可以选择。比如：

- 追踪手指的起始和终止位置（比如，把屏幕上的对象从A点移动到B点）
- 根据x、y轴坐标，追踪手指移动的方向。
- 追踪历史状态。我们可以通过调用MotionEvent的getHistorySize()方法，来获得一个手势的历史尺寸。我们可以通过移动事件的 `getHistorical<Value>` 系列函数，来获得事件之前的位置、尺寸、时间以及按压力(pressures)。当我们需要绘制用户手指痕迹时，历史状态非常有用，比如触摸绘图。查看MotionEvent来了解更多细节。
- 追踪手指在触摸屏上滑过的速度。

追踪速度

我们可以简单地用基于距离，或(和)基于手指移动方向的移动手势。但是速度经常也是追踪手势特性的一个决定性因素，甚至是判断一个手势是否发生的依据。为了让计算速度更容易，Android提供了VelocityTracker类以及**Support Library**中的VelocityTrackerCompat类。VelocityTracker类可以帮助我们追踪触摸事件中的速度因素。如果速度是手势的一个判断标准，比如快速滑动(fling)，那么这些类是很有用的。

下面是一个简单的例子，说明了VelocityTracker中API函数的用处。

```
public class MainActivity extends Activity {
    private static final String DEBUG_TAG = "Velocity";
        ...
    private VelocityTracker mVelocityTracker = null;
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        int index = event.getActionIndex();
        int action = event.getActionMasked();
        int pointerId = event.getPointerId(index);

        switch(action) {
            case MotionEvent.ACTION_DOWN:
                if(mVelocityTracker == null) {
                    // Retrieve a new VelocityTracker object to watch the velocity of
a motion.
                    mVelocityTracker = VelocityTracker.obtain();
                }
                else {
                    // Reset the velocity tracker back to its initial state.
                    mVelocityTracker.clear();
                }
                // Add a user's movement to the tracker.
                mVelocityTracker.addMovement(event);
                break;
            case MotionEvent.ACTION_MOVE:
                mVelocityTracker.addMovement(event);
                // When you want to determine the velocity, call
                // computeCurrentVelocity(). Then call getXVelocity()
                // and getYVelocity() to retrieve the velocity for each pointer ID.
                mVelocityTracker.computeCurrentVelocity(1000);
                // Log velocity of pixels per second
                // Best practice to use VelocityTrackerCompat where possible.
                Log.d("", "X velocity: " +
                        VelocityTrackerCompat.getXVelocity(mVelocityTracker,
                        pointerId));
                Log.d("", "Y velocity: " +
                        VelocityTrackerCompat.getYVelocity(mVelocityTracker,
                        pointerId));
                break;
            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_CANCEL:
                // Return a VelocityTracker object back to be re-used by others.
                mVelocityTracker.recycle();
                break;
        }
        return true;
    }
}
```

> **Note:** 需要注意的是，我们应该在ACTION_MOVE事件，而不是在ACTION_UP事件后计算速度。在ACTION_UP事件之后，计算x、y方向上的速度都会是0。

滚动手势动画

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/scroll.html

在Android中，通常使用ScrollView类来实现滚动（scroll）。任何可能超过父类边界的布局，都应该嵌套在ScrollView中，来提供一个由系统框架管理的可滚动的view。仅在某些特殊情形下，我们才要实现一个自定义scroller。本节课程就描述了这样一个情形：使用 *scrollers* 显示滚动效果，以响应触摸手势。

为了收集数据来产生滚动动画，以响应一个触摸事件，我们可以使用scrollers（Scroller或者OverScroller）。这两个类很相似，但OverScroller有一些函数，能在平移或快速滑动手势后，向用户指出已经达到内容的边缘。 `InteractiveChart` 例子使用了EdgeEffect类（实际上是EdgeEffectCompat类），在用户到达内容的边缘时显示"发光"效果。

> **Note:** 比起Scroller类，我们更推荐使用OverScroller类来产生滚动动画。OverScroller类为老设备提供了很好的向后兼容性。 另外需要注意的是，仅当我们要自己实现滚动时，才需要使用scrollers。如果我们把布局嵌套在ScrollView和HorizontalScrollView中，它们会帮我们把这些做好。

通过使用平台标准的滚动物理因素（摩擦、速度等），scroller被用来随着时间的推移产生滚动动画。实际上，scroller本身不会绘制任何东西。Scrollers只是随着时间的推移，追踪滚动的偏移量，但它们不会自动地把这些位置应用到view上。我们应该按一定频率，获取并应用这些新的坐标值，来让滚动动画更加顺滑。

理解滚动术语

在Android中，"Scrolling"这个词根据不同情景有着不同的含义。

滚动（Scrolling）是指移动视窗（viewport）（指你正在看的内容所在的'窗口'）的一般过程。当在x轴和y轴方向同时滚动时，就叫做平移（*panning*）。示例程序提供的 `InteractiveChart` 类，展示了两种不同类型的滚动，拖拽与快速滑动。

- 拖拽（dragging）是滚动的一种类型，当用户在触摸屏上拖动手指时发生。简单的拖拽一般可以通过重写 GestureDetector.OnGestureListener 的 onScroll() 来实现。关于拖拽的更多讨论，可以查看拖拽与缩放章节。
- 快速滑动（fling）这种类型的滚动，在用户快速拖拽后，抬起手指时发生。当用户抬起手指后，我们通常想继续保持滚动（移动视窗），但会一直减速直到视窗停止移动。通过重写GestureDetector.OnGestureListener的onFling()函数，使用scroller对象，可实现快速滑动。这种用法也就是本节课程的主题。

scroller对象通常会与快速滑动手势结合起来使用。但在任何我们想让UI展示滚动动画，以响应触摸事件的场景，都可以用scroller对象来实现。比如，我们可以重写onTouchEvent()函数，直接处理触摸事件，并且产生一个滚动效果或"页面对齐"动画(snapping to page)，来响应这些触摸事件。

实现基于触摸的滚动

本节讲述如何使用scroller。下面的代码段来自 `InteractiveChart` 示例。它使用GestureDetector，并且重写了GestureDetector.SimpleOnGestureListener的 `onFling()` 函数。它使用OverScroller追踪快速滑动（fling）手势。快速滑动手势后，如果用户到达内容边缘，应用会显示一种发光效果。

> **Note:** `InteractiveChart`示例程序展示了一个可缩放、平移、滑动的表格。在接下来的代码段中，`mContentRect`表示view中的一块矩形坐标区域，该区域将被用来绘制表格。在任意给定的时间点，表格中某一部分会被绘制在这个区域内。`mCurrentViewport`表示当前在屏幕上可见的那一部分表格。因为像素偏移量通常当作整型处理，所以`mContentRect`是Rect类型的。因为图表的区域范围是数值型/浮点型值，所以`mCurrentViewport`是RectF类型。

代码段的第一部分展示了 `onFling()` 函数的实现：

```
// The current viewport. This rectangle represents the currently visible
// chart domain and range. The viewport is the part of the app that the
// user manipulates via touch gestures.
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);

// The current destination rectangle (in pixel coordinates) into which the
// chart data should be drawn.
private Rect mContentRect;

private OverScroller mScroller;
private RectF mScrollerStartViewport;
...
private final GestureDetector.SimpleOnGestureListener mGestureListener
        = new GestureDetector.SimpleOnGestureListener() {
    @Override
    public boolean onDown(MotionEvent e) {
        // Initiates the decay phase of any active edge effects.
        releaseEdgeEffects();
        mScrollerStartViewport.set(mCurrentViewport);
        // Aborts any active scroll animations and invalidates.
        mScroller.forceFinished(true);
        ViewCompat.postInvalidateOnAnimation(InteractiveLineGraphView.this);
        return true;
    }
    ...
```

```
    @Override
    public boolean onFling(MotionEvent e1, MotionEvent e2,
            float velocityX, float velocityY) {
        fling((int) -velocityX, (int) -velocityY);
        return true;
    }
};

private void fling(int velocityX, int velocityY) {
    // Initiates the decay phase of any active edge effects.
    releaseEdgeEffects();
    // Flings use math in pixels (as opposed to math based on the viewport).
    Point surfaceSize = computeScrollSurfaceSize();
    mScrollerStartViewport.set(mCurrentViewport);
    int startX = (int) (surfaceSize.x * (mScrollerStartViewport.left -
            AXIS_X_MIN) / (
            AXIS_X_MAX - AXIS_X_MIN));
    int startY = (int) (surfaceSize.y * (AXIS_Y_MAX -
            mScrollerStartViewport.bottom) / (
            AXIS_Y_MAX - AXIS_Y_MIN));
    // Before flinging, aborts the current animation.
    mScroller.forceFinished(true);
    // Begins the animation
    mScroller.fling(
            // Current scroll position
            startX,
            startY,
            velocityX,
            velocityY,
            /*
             * Minimum and maximum scroll positions. The minimum scroll
             * position is generally zero and the maximum scroll position
             * is generally the content size less the screen size. So if the
             * content width is 1000 pixels and the screen width is 200
             * pixels, the maximum scroll offset should be 800 pixels.
             */
            0, surfaceSize.x - mContentRect.width(),
            0, surfaceSize.y - mContentRect.height(),
            // The edges of the content. This comes into play when using
            // the EdgeEffect class to draw "glow" overlays.
            mContentRect.width() / 2,
            mContentRect.height() / 2);
    // Invalidates to trigger computeScroll()
    ViewCompat.postInvalidateOnAnimation(this);
}
```

当 `onFling()` 函数调用postInvalidateOnAnimation()时，它会触发computeScroll()来更新x、y的值。通常一个子view用scroller对象来产生滚动动画时会这样做，就像本例一样。

大多数views直接通过scrollTo()函数传递scroller对象的x、y坐标值。接下来的 `computeScroll()` 函数的实现中采用了一种不同的方式。它调用computeScrollOffset()函数来获得当前位置的x、y值。当满足边缘显示发光效果的条件时（图表已被放大显示，x或y值超过边界，并且app当前没有显示overscroll），这段代码会设置overscroll发光效果，并调用 `postInvalidateOnAnimation()` 函数来让view失效重绘：

```
// Edge effect / overscroll tracking objects.
private EdgeEffectCompat mEdgeEffectTop;
private EdgeEffectCompat mEdgeEffectBottom;
private EdgeEffectCompat mEdgeEffectLeft;
private EdgeEffectCompat mEdgeEffectRight;

private boolean mEdgeEffectTopActive;
private boolean mEdgeEffectBottomActive;
private boolean mEdgeEffectLeftActive;
private boolean mEdgeEffectRightActive;

@Override
public void computeScroll() {
    super.computeScroll();

    boolean needsInvalidate = false;

    // The scroller isn't finished, meaning a fling or programmatic pan
    // operation is currently active.
    if (mScroller.computeScrollOffset()) {
        Point surfaceSize = computeScrollSurfaceSize();
        int currX = mScroller.getCurrX();
        int currY = mScroller.getCurrY();

        boolean canScrollX = (mCurrentViewport.left > AXIS_X_MIN
                || mCurrentViewport.right < AXIS_X_MAX);
        boolean canScrollY = (mCurrentViewport.top > AXIS_Y_MIN
                || mCurrentViewport.bottom < AXIS_Y_MAX);

        /*
         * If you are zoomed in and currX or currY is
         * outside of bounds and you're not already
         * showing overscroll, then render the overscroll
         * glow edge effect.
         */
        if (canScrollX
                && currX < 0
                && mEdgeEffectLeft.isFinished()
                && !mEdgeEffectLeftActive) {
            mEdgeEffectLeft.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectLeftActive = true;
            needsInvalidate = true;
        } else if (canScrollX
                && currX > (surfaceSize.x - mContentRect.width())
```

```
                && mEdgeEffectRight.isFinished()
                && !mEdgeEffectRightActive) {
            mEdgeEffectRight.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectRightActive = true;
            needsInvalidate = true;
        }

        if (canScrollY
                && currY < 0
                && mEdgeEffectTop.isFinished()
                && !mEdgeEffectTopActive) {
            mEdgeEffectTop.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectTopActive = true;
            needsInvalidate = true;
        } else if (canScrollY
                && currY > (surfaceSize.y - mContentRect.height())
                && mEdgeEffectBottom.isFinished()
                && !mEdgeEffectBottomActive) {
            mEdgeEffectBottom.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectBottomActive = true;
            needsInvalidate = true;
        }
        ...
    }
```

这是缩放部分的代码：

```
// Custom object that is functionally similar to Scroller
Zoomer mZoomer;
private PointF mZoomFocalPoint = new PointF();
...

// If a zoom is in progress (either programmatically or via double
// touch), performs the zoom.
if (mZoomer.computeZoom()) {
    float newWidth = (1f - mZoomer.getCurrZoom()) *
            mScrollerStartViewport.width();
    float newHeight = (1f - mZoomer.getCurrZoom()) *
            mScrollerStartViewport.height();
    float pointWithinViewportX = (mZoomFocalPoint.x -
            mScrollerStartViewport.left)
            / mScrollerStartViewport.width();
    float pointWithinViewportY = (mZoomFocalPoint.y -
            mScrollerStartViewport.top)
            / mScrollerStartViewport.height();
    mCurrentViewport.set(
            mZoomFocalPoint.x - newWidth * pointWithinViewportX,
            mZoomFocalPoint.y - newHeight * pointWithinViewportY,
            mZoomFocalPoint.x + newWidth * (1 - pointWithinViewportX),
            mZoomFocalPoint.y + newHeight * (1 - pointWithinViewportY));
    constrainViewport();
    needsInvalidate = true;
}
if (needsInvalidate) {
    ViewCompat.postInvalidateOnAnimation(this);
}
```

这是上面代码段中调用过的 `computeScrollSurfaceSize()` 函数。它会以像素为单位计算当前可滚动的尺寸。举例来说，如果整个图表区域都是可见的，它的值就简单地等于 `mContentRect` 的大小。如果图表在两个方向上都放大到200%，此函数返回的尺寸在水平、垂直方向上都会大两倍。

```
private Point computeScrollSurfaceSize() {
    return new Point(
            (int) (mContentRect.width() * (AXIS_X_MAX - AXIS_X_MIN)
                    / mCurrentViewport.width()),
            (int) (mContentRect.height() * (AXIS_Y_MAX - AXIS_Y_MIN)
                    / mCurrentViewport.height()));
}
```

关于scroller用法的另一个示例，可查看ViewPager类的源代码。它用滚动来响应快速滑动（fling），并且使用滚动来实现“页面对齐”(snapping to page)动画。

处理多点触控手势

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/multi.html

多点触控手势是指在同一时间有多点（手指）触碰屏幕。本节课程讲述，如何检测涉及多点的触摸手势。

追踪多点

当多个手指同时触摸屏幕时，系统会产生如下的触摸事件：

- ACTION_DOWN - 针对触摸屏幕的第一个点。此事件是手势的开端。第一触摸点的数据在MotionEvent中的索引总是0。
- ACTION_POINTER_DOWN - 针对第一点后，出现在屏幕上额外的点。这个点的数据在MotionEvent中的索引，可以通过getActionIndex()获得。
- ACTION_MOVE - 在按下手势期间发生变化。
- ACTION_POINTER_UP - 当非主要点（non-primary pointer）离开屏幕时，发送此事件。
- ACTION_UP - 当最后一点离开屏幕时发送此事件。

我们可以通过各个点的索引以及id，单独地追踪MotionEvent中的每个点。

- **Index**：MotionEvent把各个点的信息都存储在一个数组中。点的索引值就是它在数组中的位置。大多数用来与点交互的MotionEvent函数都是以索引值而不是点的ID作为参数的。
- **ID**：每个点也都有一个ID映射，该ID映射在整个手势期间一直存在，以便我们单独地追踪每个点。

每个独立的点在移动事件中出现的次序是不固定的。因此，从一个事件到另一个事件，点的索引值是可以改变的，但点的ID在它的生命周期内是保证不会改变的。使用getPointerId()可以获得一个点的ID，在手势随后的移动事件中，就可以用该ID来追踪这个点。对于随后一系列的事件，可以使用findPointerIndex()函数，来获得对应给定ID的点在移动事件中的索引值。如下：

```
private int mActivePointerId;

public boolean onTouchEvent(MotionEvent event) {
    ....
    // Get the pointer ID
    mActivePointerId = event.getPointerId(0);

    // ... Many touch events later...

    // Use the pointer ID to find the index of the active pointer
    // and fetch its position
    int pointerIndex = event.findPointerIndex(mActivePointerId);
    // Get the pointer's current position
    float x = event.getX(pointerIndex);
    float y = event.getY(pointerIndex);
}
```

获取MotionEvent的动作

我们应该总是使用getActionMasked()函数（或者用MotionEventCompat.getActionMasked()这个兼容版本更好）来获取MotionEvent的动作(action)。与旧的getAction()函数不同的是，`getActionMasked()` 是设计用来处理多点触摸的。它会返回执行过的动作的掩码值，不包括点的索引位。然后，我们可以使用 `getActionIndex()` 来获得与该动作关联的点的索引值。这在接下来的代码段中可以看到。

> **Note:** 这个样例使用的是MotionEventCompat类。该类在**Support Library**中。我们应该使用MotionEventCompat类，来提供对更多平台的支持。需要注意的一点是，MotionEventCompat并不是MotionEvent类的替代品。准确来说，它提供了一些静态工具类函数，我们可以把MotionEvent对象作为参数传递给这些函数，来得到与事件相关的动作。

```
int action = MotionEventCompat.getActionMasked(event);
// Get the index of the pointer associated with the action.
int index = MotionEventCompat.getActionIndex(event);
int xPos = -1;
int yPos = -1;

Log.d(DEBUG_TAG,"The action is " + actionToString(action));

if (event.getPointerCount() > 1) {
    Log.d(DEBUG_TAG,"Multitouch event");
    // The coordinates of the current screen contact, relative to
    // the responding View or Activity.
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);

} else {
    // Single touch event
    Log.d(DEBUG_TAG,"Single touch event");
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);
}
...

// Given an action int, returns a string description
public static String actionToString(int action) {
    switch (action) {

        case MotionEvent.ACTION_DOWN: return "Down";
        case MotionEvent.ACTION_MOVE: return "Move";
        case MotionEvent.ACTION_POINTER_DOWN: return "Pointer Down";
        case MotionEvent.ACTION_UP: return "Up";
        case MotionEvent.ACTION_POINTER_UP: return "Pointer Up";
        case MotionEvent.ACTION_OUTSIDE: return "Outside";
        case MotionEvent.ACTION_CANCEL: return "Cancel";
    }
    return "";
}
```

关于多点触摸的更多内容以及示例，可以查看拖拽与缩放章节。

拖拽与缩放

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/scale.html

本节课程讲述，使用onTouchEvent()截获触摸事件后，如何使用触摸手势拖拽、缩放屏幕上的对象。

拖拽一个对象

> 如果我们的目标版本为3.0或以上，我们可以使用View.OnDragListener监听内置的拖放（drag-and-drop）事件，拖拽与释放中有更多相关描述。

对于触摸手势来说，一个很常见的操作是在屏幕上拖拽一个对象。接下来的代码段让用户可以拖拽屏幕上的图片。需要注意以下几点：

- 拖拽操作时，即使有额外的手指放置到屏幕上了，app也必须保持对最初的点（手指）的追踪。比如，想象在拖拽图片时，用户放置了第二根手指在屏幕上，并且抬起了第一根手指。如果我们的app只是单独地追踪每个点，它会把第二个点当做默认的点，并且把图片移到该点的位置。
- 为了防止这种情况发生，我们的app需要区分初始点以及随后任意的触摸点。要做到这一点，它需要追踪处理多触摸手势章节中提到过的 ACTION_POINTER_DOWN 和 ACTION_POINTER_UP 事件。每当第二根手指按下或拿起时，ACTION_POINTER_DOWN 和 ACTION_POINTER_UP 事件就会传递给 `onTouchEvent()` 回调函数。
- 当ACTION_POINTER_UP事件发生时，示例程序会移除对该点的索引值的引用，确保操作中的点的ID(the active pointer ID)不会引用已经不在触摸屏上的触摸点。这种情况下，app会选择另一个触摸点来作为操作中(active)的点，并保存它当前的x、y值。由于在ACTION_MOVE事件时，这个保存的位置会被用来计算屏幕上的对象将要移动的距离，所以app会始终根据正确的触摸点来计算移动的距离。

下面的代码段允许用户拖拽屏幕上的对象。它会记录操作中的点（active pointer）的初始位置，计算触摸点移动过的距离，再把对象移动到新的位置。如上所述，它也正确地处理了额外触摸点的可能。

需要注意的是，代码段中使用了getActionMasked()函数。我们应该始终使用这个函数（或者最好用MotionEventCompat.getActionMasked()这个兼容版本）来获得MotionEvent对应的动作(action)。不像旧的getAction()函数，`getActionMasked()` 就是设计用来处理多点触摸的。它会返回执行过的动作的掩码值，不包括该点的索引位。

```
// The ‘active pointer’ is the one currently moving our object.
```

```
private int mActivePointerId = INVALID_POINTER_ID;

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);

    final int action = MotionEventCompat.getActionMasked(ev);

    switch (action) {
    case MotionEvent.ACTION_DOWN: {
        final int pointerIndex = MotionEventCompat.getActionIndex(ev);
        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);

        // Remember where we started (for dragging)
        mLastTouchX = x;
        mLastTouchY = y;
        // Save the ID of this pointer (for dragging)
        mActivePointerId = MotionEventCompat.getPointerId(ev, 0);
        break;
    }

    case MotionEvent.ACTION_MOVE: {
        // Find the index of the active pointer and fetch its position
        final int pointerIndex =
                MotionEventCompat.findPointerIndex(ev, mActivePointerId);

        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);

        // Calculate the distance moved
        final float dx = x - mLastTouchX;
        final float dy = y - mLastTouchY;

        mPosX += dx;
        mPosY += dy;

        invalidate();

        // Remember this touch position for the next move event
        mLastTouchX = x;
        mLastTouchY = y;

        break;
    }

    case MotionEvent.ACTION_UP: {
        mActivePointerId = INVALID_POINTER_ID;
        break;
    }

    case MotionEvent.ACTION_CANCEL: {
```

```
            mActivePointerId = INVALID_POINTER_ID;
            break;
        }

        case MotionEvent.ACTION_POINTER_UP: {

            final int pointerIndex = MotionEventCompat.getActionIndex(ev);
            final int pointerId = MotionEventCompat.getPointerId(ev, pointerIndex);

            if (pointerId == mActivePointerId) {
                // This was our active pointer going up. Choose a new
                // active pointer and adjust accordingly.
                final int newPointerIndex = pointerIndex == 0 ? 1 : 0;
                mLastTouchX = MotionEventCompat.getX(ev, newPointerIndex);
                mLastTouchY = MotionEventCompat.getY(ev, newPointerIndex);
                mActivePointerId = MotionEventCompat.getPointerId(ev, newPointerIndex);
            }
            break;
        }
        }
        return true;
    }
```

通过拖拽平移

前一节展示了一个，在屏幕上拖拽对象的例子。另一个常见的场景是平移（*panning*），平移是指用户通过拖拽移动引起x、y轴方向发生滚动(scrolling)。上面的代码段直接截获了MotionEvent动作来实现拖拽。这一部分的代码段，利用了平台对常用手势的内置支持。它重写了GestureDetector.SimpleOnGestureListener的onScroll()函数。

更详细地说，当用户拖拽手指来平移内容时， `onScroll()` 函数就会被调用。 `onScroll()` 函数只会在手指按下的情况下被调用，一旦手指离开屏幕了，要么手势终止，要么快速滑动(fling)手势开始（如果手指在离开屏幕前快速移动了一段距离）。关于滚动与快速滑动的更多讨论，可以查看滚动手势动画章节。

这里是 `onScroll()` 的相关代码段：

```
// The current viewport. This rectangle represents the currently visible
// chart domain and range.
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);

// The current destination rectangle (in pixel coordinates) into which the
// chart data should be drawn.
private Rect mContentRect;

private final GestureDetector.SimpleOnGestureListener mGestureListener
            = new GestureDetector.SimpleOnGestureListener() {
...

@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
            float distanceX, float distanceY) {
    // Scrolling uses math based on the viewport (as opposed to math using pixels).

    // Pixel offset is the offset in screen pixels, while viewport offset is the
    // offset within the current viewport.
    float viewportOffsetX = distanceX * mCurrentViewport.width()
            / mContentRect.width();
    float viewportOffsetY = -distanceY * mCurrentViewport.height()
            / mContentRect.height();
    ...
    // Updates the viewport, refreshes the display.
    setViewportBottomLeft(
            mCurrentViewport.left + viewportOffsetX,
            mCurrentViewport.bottom + viewportOffsetY);
    ...
    return true;
}
```

`onScroll()` 函数中滑动视窗(viewport)来响应触摸手势的实现：

```
/**
 * Sets the current viewport (defined by mCurrentViewport) to the given
 * X and Y positions. Note that the Y value represents the topmost pixel position,
 * and thus the bottom of the mCurrentViewport rectangle.
 */
private void setViewportBottomLeft(float x, float y) {
    /*
     * Constrains within the scroll range. The scroll range is simply the viewport
     * extremes (AXIS_X_MAX, etc.) minus the viewport size. For example, if the
     * extremes were 0 and 10, and the viewport size was 2, the scroll range would
     * be 0 to 8.
     */

    float curWidth = mCurrentViewport.width();
    float curHeight = mCurrentViewport.height();
    x = Math.max(AXIS_X_MIN, Math.min(x, AXIS_X_MAX - curWidth));
    y = Math.max(AXIS_Y_MIN + curHeight, Math.min(y, AXIS_Y_MAX));

    mCurrentViewport.set(x, y - curHeight, x + curWidth, y);

    // Invalidates the View to update the display.
    ViewCompat.postInvalidateOnAnimation(this);
}
```

使用触摸手势进行缩放

如同检测常用手势章节中提到的，GestureDetector可以帮助我们检测Android中的常见手势，例如滚动，快速滚动以及长按。对于缩放，Android也提供了ScaleGestureDetector类。当我们想让view能识别额外的手势时，我们可以同时使用GestureDetector和ScaleGestureDetector类。

为了报告检测到的手势事件，手势检测需要一个作为构造函数参数的listener对象。ScaleGestureDetector使用ScaleGestureDetector.OnScaleGestureListener。Android提供了ScaleGestureDetector.SimpleOnScaleGestureListener类作为帮助类，如果我们不是关注所有的手势事件，我们可以继承(extend)它。

基本的缩放示例

下面的代码段展示了缩放功能中的基本部分。

```
private ScaleGestureDetector mScaleDetector;
private float mScaleFactor = 1.f;

public MyCustomView(Context mContext){
    ...
    // View code goes here
    ...
    mScaleDetector = new ScaleGestureDetector(context, new ScaleListener());
}

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);
    return true;
}

@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    canvas.save();
    canvas.scale(mScaleFactor, mScaleFactor);
    ...
    // onDraw() code goes here
    ...
    canvas.restore();
}

private class ScaleListener
        extends ScaleGestureDetector.SimpleOnScaleGestureListener {
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        mScaleFactor *= detector.getScaleFactor();

        // Don't let the object get too small or too large.
        mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

        invalidate();
        return true;
    }
}
```

更加复杂的缩放示例

这是本章节提供的 `InteractiveChart` 示例中一个更复杂的示范。通过使用ScaleGestureDetector中的"span"(getCurrentSpanX/Y)和"focus"(getFocusX/Y)功能，`InteractiveChart` 示例同时支持滚动（平移）以及多指缩放。

```
@Override
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);
private Rect mContentRect;
private ScaleGestureDetector mScaleGestureDetector;
...
public boolean onTouchEvent(MotionEvent event) {
    boolean retVal = mScaleGestureDetector.onTouchEvent(event);
    retVal = mGestureDetector.onTouchEvent(event) || retVal;
    return retVal || super.onTouchEvent(event);
}

/**
 * The scale listener, used for handling multi-finger scale gestures.
 */
private final ScaleGestureDetector.OnScaleGestureListener mScaleGestureListener
        = new ScaleGestureDetector.SimpleOnScaleGestureListener() {
    /**
     * This is the active focal point in terms of the viewport. Could be a local
     * variable but kept here to minimize per-frame allocations.
     */
    private PointF viewportFocus = new PointF();
    private float lastSpanX;
    private float lastSpanY;

    // Detects that new pointers are going down.
    @Override
    public boolean onScaleBegin(ScaleGestureDetector scaleGestureDetector) {
        lastSpanX = ScaleGestureDetectorCompat.
                getCurrentSpanX(scaleGestureDetector);
        lastSpanY = ScaleGestureDetectorCompat.
                getCurrentSpanY(scaleGestureDetector);
        return true;
    }

    @Override
    public boolean onScale(ScaleGestureDetector scaleGestureDetector) {

        float spanX = ScaleGestureDetectorCompat.
                getCurrentSpanX(scaleGestureDetector);
        float spanY = ScaleGestureDetectorCompat.
                getCurrentSpanY(scaleGestureDetector);

        float newWidth = lastSpanX / spanX * mCurrentViewport.width();
        float newHeight = lastSpanY / spanY * mCurrentViewport.height();

        float focusX = scaleGestureDetector.getFocusX();
        float focusY = scaleGestureDetector.getFocusY();
        // Makes sure that the chart point is within the chart region.
        // See the sample for the implementation of hitTest().
        hitTest(scaleGestureDetector.getFocusX(),
                scaleGestureDetector.getFocusY(),
```

```
                viewportFocus);

        mCurrentViewport.set(
                viewportFocus.x
                        - newWidth * (focusX - mContentRect.left)
                        / mContentRect.width(),
                viewportFocus.y
                        - newHeight * (mContentRect.bottom - focusY)
                        / mContentRect.height(),
                0,
                0);
        mCurrentViewport.right = mCurrentViewport.left + newWidth;
        mCurrentViewport.bottom = mCurrentViewport.top + newHeight;
        ...
        // Invalidates the View to update the display.
        ViewCompat.postInvalidateOnAnimation(InteractiveLineGraphView.this);

        lastSpanX = spanX;
        lastSpanY = spanY;
        return true;
    }
};
```

管理ViewGroup中的触摸事件

> 编写:Andrwyw - 原文:http://developer.android.com/training/gestures/viewgroup.html

因为很多时候是用ViewGroup的子类来做不同触摸事件的目标，而不是ViewGroup本身，所以处理ViewGroup中的触摸事件需要特别注意。 为了确保每个view能正确地接收到它们想要的触摸事件，可以重写onInterceptTouchEvent()函数。

在ViewGroup中截获触摸事件

每当在ViewGroup（包括它的子View）的表面上检测到一个触摸事件，onInterceptTouchEvent()都会被调用。如果 `onInterceptTouchEvent()` 返回 `true` ，MotionEvent就被截获了，这表示它不会被传递给其子View，而是传递给该父view自身的onTouchEvent()方法。

`onInterceptTouchEvent()` 方法让父view能够在它的子view之前处理触摸事件。如果我们让 `onInterceptTouchEvent()` 返回 `true` ，则之前处理触摸事件的子view会收到ACTION_CANCEL事件，并且该点之后的事件会被发送给该父view自身的 `onTouchEvent()` 函数，进行常规处理。 `onInterceptTouchEvent()` 也可以返回 `false` ，这样事件沿view层级分发到目标前，父view可以简单地观察该事件。这里的目标是指，通过 `onTouchEvent()` 处理消息事件的view。

接下来的代码段中， `MyViewGroup` 继承自ViewGroup。 `MyViewGroup` 有多个子view。如果我们在某个子View上水平地拖动手指，该子view不会接收到触摸事件，而是应该由 `MyViewGroup` 处理这些触摸事件来滚动它的内容。然而，如果我们点击子view中的button，或垂直地滚动子view，则父view不会截获这些触摸事件，因为子view本身就是预定目标。在这些情况下， `onInterceptTouchEvent()` 应该返回 `false` ， `MyViewGroup` 的 `onTouchEvent()` 也不会被调用。

```
public class MyViewGroup extends ViewGroup {

    private int mTouchSlop;

    ...

    ViewConfiguration vc = ViewConfiguration.get(view.getContext());
    mTouchSlop = vc.getScaledTouchSlop();

    ...

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
```

```
        /*
         * This method JUST determines whether we want to intercept the motion.
         * If we return true, onTouchEvent will be called and we do the actual
         * scrolling there.
         */


        final int action = MotionEventCompat.getActionMasked(ev);

        // Always handle the case of the touch gesture being complete.
        if (action == MotionEvent.ACTION_CANCEL || action == MotionEvent.ACTION_UP) {
            // Release the scroll.
            mIsScrolling = false;
            return false; // Do not intercept touch event, let the child handle it
        }

        switch (action) {
            case MotionEvent.ACTION_MOVE: {
                if (mIsScrolling) {
                    // We're currently scrolling, so yes, intercept the
                    // touch event!
                    return true;
                }

                // If the user has dragged her finger horizontally more than
                // the touch slop, start the scroll

                // left as an exercise for the reader
                final int xDiff = calculateDistanceX(ev);

                // Touch slop should be calculated using ViewConfiguration
                // constants.
                if (xDiff > mTouchSlop) {
                    // Start scrolling!
                    mIsScrolling = true;
                    return true;
                }
                break;
            }
            ...
        }

        // In general, we don't want to intercept touch events. They should be
        // handled by the child view.
        return false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        // Here we actually handle the touch event (e.g. if the action is ACTION_MOVE,
        // scroll this container).
        // This method will only be called if the touch event was intercepted in
        // onInterceptTouchEvent
```

```
        ...
    }
}
```

注意ViewGroup也提供了requestDisallowInterceptTouchEvent()方法。当子view不想该父view和祖先view通过 `onInterceptTouchEvent()` 截获它的触摸事件时，可调用ViewGroup的该方法。

使用ViewConfiguration的常量

上面的代码段中使用了当前的ViewConfiguration来初始化 `mTouchSlop` 变量。我们可以使用ViewConfiguration类来获取Android系统常用的一些距离、速度、时间值。

“Touch slop”是指在被识别为移动的手势前，用户触摸可移动的那一段像素距离。Touch slop通常用来预防用户在做一些其他触摸操作时，出现意外地滑动，例如触摸屏幕上的组件。

另外两个常用的ViewConfiguration函数是getScaledMinimumFlingVelocity()和getScaledMaximumFlingVelocity()。这两个函数会返回初始化一个快速滑动(fling)的最小、最大速度（分别地），以像素每秒为测量单位。如：

```
ViewConfiguration vc = ViewConfiguration.get(view.getContext());
private int mSlop = vc.getScaledTouchSlop();
private int mMinFlingVelocity = vc.getScaledMinimumFlingVelocity();
private int mMaxFlingVelocity = vc.getScaledMaximumFlingVelocity();

...

case MotionEvent.ACTION_MOVE: {
    ...
    float deltaX = motionEvent.getRawX() - mDownX;
    if (Math.abs(deltaX) > mSlop) {
        // A swipe occurred, do something
    }

...

case MotionEvent.ACTION_UP: {
    ...
    } if (mMinFlingVelocity <= velocityX && velocityX <= mMaxFlingVelocity
            && velocityY < velocityX) {
        // The criteria have been satisfied, do something
    }
}
```

扩展子view的可触摸区域

Android提供了TouchDelegate类，让父view扩展超出子view自身边界的可触摸区域。这在当子view很小，但需要一个更大的触摸区域时非常有用。如果需要，我们也可以使用这种方式来实现对子view的触摸区域的收缩。

在下面的例子中，ImageButton对象是所谓的"delegate view"（是指触摸区域将被父view扩展的那个子view）。这是布局文件：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
     android:id="@+id/parent_layout"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context=".MainActivity" >

     <ImageButton android:id="@+id/button"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:background="@null"
          android:src="@drawable/icon" />
</RelativeLayout>
```

下面的代码段做了这样几件事：

- 获得父view对象并发送一个Runnable到UI线程。这会确保父view在调用getHitRect()函数前会布局它的子view。`getHitRect()` 函数会获得子view在父view坐标系中的点击矩形（触摸区域）。
- 找到ImageButton子view，然后调用 `getHitRect()` 来获得它的触摸区域的边界。
- 扩展ImageButton的点击矩形的边界。
- 实例化一个TouchDelegate对象，并把扩展过的点击矩形和ImageButton子view作为参数传递给它。
- 设置父view的TouchDelegate，这样在touch delegate边界内的点击就会传递到该子view上。

在ImageButton子view的touch delegate范围内，父view会接收到所有的触摸事件。如果触摸事件发生在子view自身的点击矩形中，父view会把触摸事件交给子view处理。

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Get the parent view
        View parentView = findViewById(R.id.parent_layout);

        parentView.post(new Runnable() {
            // Post in the parent's message queue to make sure the parent
            // lays out its children before you call getHitRect()
```

```
                @Override
                public void run() {
                    // The bounds for the delegate view (an ImageButton
                    // in this example)
                    Rect delegateArea = new Rect();
                    ImageButton myButton = (ImageButton) findViewById(R.id.button);
                    myButton.setEnabled(true);
                    myButton.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View view) {
                            Toast.makeText(MainActivity.this,
                                    "Touch occurred within ImageButton touch region.",
                                    Toast.LENGTH_SHORT).show();
                        }
                    });

                    // The hit rectangle for the ImageButton
                    myButton.getHitRect(delegateArea);

                    // Extend the touch area of the ImageButton beyond its bounds
                    // on the right and bottom.
                    delegateArea.right += 100;
                    delegateArea.bottom += 100;

                    // Instantiate a TouchDelegate.
                    // "delegateArea" is the bounds in local coordinates of
                    // the containing view to be mapped to the delegate view.
                    // "myButton" is the child view that should receive motion
                    // events.
                    TouchDelegate touchDelegate = new TouchDelegate(delegateArea,
                            myButton);

                    // Sets the TouchDelegate on the parent view, such that touches
                    // within the touch delegate bounds are routed to the child.
                    if (View.class.isInstance(myButton.getParent())) {
                        ((View) myButton.getParent()).setTouchDelegate(touchDelegate);
                    }
                }
            });
        }
    }
```

处理键盘输入

> 编写:zhaochunqi - 原文:http://developer.android.com/training/keyboard-input/index.html

当当前焦点在 UI 的文本框上时，Android 系统会在屏幕上显示一个键盘 — 被称为软输入法。为了提供最好的用户体验，我们可以指定我们期望的输入类型的特征（例如，是否是电话号码或Email地址）和输入法的表现形式（例如，是否需要自动纠正拼写错误）。

除了使用屏幕上的输入法，Android也支持实体键盘，所以充分利用可能会被用户接入的外接键盘来优化用户的交互体验是很重要的。

接下来的课程会讨论上述这些主题和更多相关内容。

Lessons

指定输入法类型

学习如何表现特定的软输入法，如为电话号码、网址和其他一些格式所做的设计。同样应该学习如何指定一些属性，例如拼写建议和像确定**(Done)**或者下一步**(Next)**这样的动作按钮。

处理输入法的显示

学习如何合适地展示软输入法，和如何让我们的布局作出调整，来适合因为输入法而减少的屏幕空间。

支持键盘导航

学习如何验证用户是否能够使用键盘导航我们的应用以及如何对导航顺序做出必要的改变。

处理键盘行为

学习如何对用户的键盘输入作出回应。

指定输入法类型

> 编写:zhaochunqi - 原文:http://developer.android.com/training/keyboard-input/style.html

每个文本框都对应特定类型的文本输入，如Email地址，电话号码，或者纯文本。为应用中的每一个文本框指定输入类型是很重要的，这样做可以让系统展示更为合适的软输入法（比如虚拟键盘）。

除了输入法可用的按钮类型之外，我们还应该指定一些行为，例如，输入法是否提供拼写建议，新的句子首字母大写，和将回车按钮替换成动作按钮（如 **Done** 或者 **Next**）。这节课介绍了如何添加这些属性。

指定键盘类型

通过将 android:inputType 属性添加到 <EditText> 节点中，我们可以为文本框声明输入法。

举例来说，如果我们想要一个用于输入电话号码的输入法，那么使用 `"phone"` 值：

```
<EditText
    android:id="@+id/phone"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/phone_hint"
    android:inputType="phone" />
```



Figure 1. `phone` 输入类型

或者如果文本框用于输入密码，那么使用 `"textPassword"` 值来隐藏用户的输入：

```
<EditText
    android:id="@+id/password"
    android:hint="@string/password_hint"
    android:inputType="textPassword"
    ... />
```

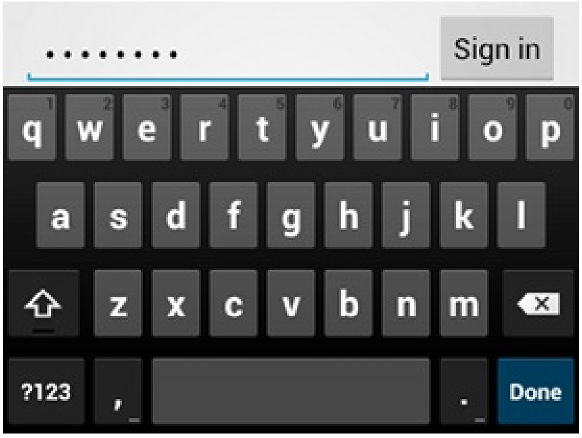


Figure 2. `textPassword` 输入类型

有几种可供选择的值在 `android:inputType` 记录在属性中，一些值可以组合起来实现特定的输入法外观和附加的行为。

开启拼写建议和其它行为

android:inputType 属性允许我们为输入法指定不同的行为。最为重要的是，如果文本框用于基本的文本输入（如短信息），那么我们应该使用 `"textAutoCorrect"` 值来开启自动拼写修正。

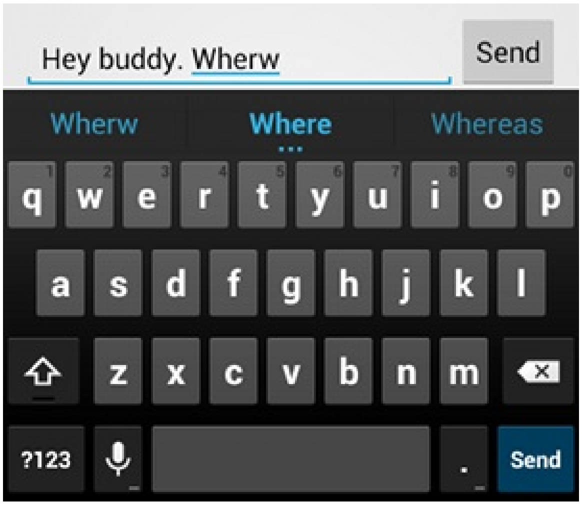


Figure 3. 添加 `textAutoCorrect` 为拼写错误提供自动修正

我们可以将不同的行为和输入法形式组合到 android:inputType 这个属性。如：如何创建一个文本框，里面的句子首字母大写并开启拼写修正：

```
<EditText
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType=
        "textCapSentences|textAutoCorrect"
    ... />
```

指定输入法的行为

多数的软键盘会在底部角落里为用户提供一个合适的动作按钮来触发当前文本框的操作。默认情况下，系统使用 **Next** 或者 **Done**，除非我们的文本框允许多行文本（如 `android:inputType="textMultiLine"` ），这种情况下，动作按钮就是回车换行。然而，我们可以指定一些更适合我们文本框的额外动作，比如 **Send** 和 **Go**。



Figure 4. 当我们声明了 `android:imeOptions="actionSend"` ，会出现 Send 按钮。

使用android:imeOptions 属性，并设置一个动作值（如 `"actionSend"` 或 `"actionSearch"` ），来指定键盘的动作按钮。如：

```
<EditText
    android:id="@+id/search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/search_hint"
    android:inputType="text"
    android:imeOptions="actionSend" />
```

然后，我们可以通过为 EditText 节点定义 TextView.OnEditorActionListener 来监听动作按钮的按压。在监听器中，响应 EditorInfo 类中定义的适合的 IME action ID，如 IME_ACTION_SEND 。例如:

```
EditText editText = (EditText) findViewById(R.id.search);
editText.setOnEditorActionListener(new OnEditorActionListener() {
    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
        boolean handled = false;
        if (actionId == EditorInfo.IME_ACTION_SEND) {
            sendMessage();
            handled = true;
        }
        return handled;
    }
});
```

处理输入法可见性

> 编写:zhaochunqi - 原文:http://developer.android.com/training/keyboard-input/visibility.html

当输入焦点移入或移出可编辑的文本框时，Android会相应的显示或隐藏输入法（如虚拟键盘）。系统也会决定输入法上方的 UI 和文本框的显示方式。举例来说，当屏幕上垂直空间被压缩时，文本框可能填充输入法上方所有的空间。对于多数的应用来说，这些默认的行为基本就足够了。

然而，在一些事例中，我们可能会想要更加直接地控制输入法的显示，指定在输入法显示的时候，如何显示我们的布局。这节课会解释如何控制和响应输入法的可见性。

在Activity启动时显示输入法

尽管Android会在Activity启动时将焦点放在布局中的第一个文本框，但是并不会显示输入法。因为输入文本可能并不是activity中的首要任务，所以不显示输入法是很合理的。可是，如果输入文本确实是首要的任务（如在登录界面中），那么可能需要默认显示输入法。

为了在activity启动时显示输入法，添加 android:windowSoftInputMode 属性到 <activity> 节点中，并将该属性的值设为 `"stateVisible"` 。如下：

```
<application ... >
    <activity
        android:windowSoftInputMode="stateVisible" ... >
        ...
    </activity>
    ...
</application>
```

> **Note:** 如果用户的设备有一个实体键盘，那么不会显示软输入法。

根据需要显示输入法

如果我们想要确保输入法在activity生命周期的某个方法中是可见的，那么可以使用 InputMethodManager 来实现。

举例来说，下面的方法调用了一个需要用户填写文本的View，调用了 requestFocus() 来获取焦点，然后调用 showSoftInput() 来打开输入法。

```
public void showSoftKeyboard(View view) {
    if (view.requestFocus()) {
        InputMethodManager imm = (InputMethodManager)
                getSystemService(Context.INPUT_METHOD_SERVICE);
        imm.showSoftInput(view, InputMethodManager.SHOW_IMPLICIT);
    }
}
```

> **Note:** 一旦输入法可见，我们不应该以编程的方式来隐藏它。系统会在用户结束文本框的任务时隐藏输入法，或者可以使用系统控制（如返回键）来隐藏。

指定 UI 的响应方式

当输入法显示在屏幕上时，会减少 app UI 中的可用空间。系统会决定如何调整 UI 可见的部分，但是这样做不一定正确。为了确保应用的最佳表现，我们应该在 UI 的剩余空间中展示我们想要展示的系统界面。

为了在activity中声明合适的处理方法，可以在 manifest 文件的 <activity> 节点中使用 android:windowSoftInputMode 属性，并将该属性的值设为"adjust"。

举例来说，为了确保系统会在可用空间中重新调整布局的大小——确保所有的布局内容都可以被使用（尽管可能需要滑动）——使用 `"adjustResize"`：

```
<application ... >
    <activity
        android:windowSoftInputMode="adjustResize" ... >
        ...
    </activity>
    ...
</application>
```

我们可以结合上述调整说明和初始化输入法可见性说明：

```
    <activity
        android:windowSoftInputMode="stateVisible|adjustResize" ... >
        ...
    </activity>
```

如果 UI 中包含用户可能需要在文本输入时立即执行的事情，那么使用 `"adjustResize"` 是很重要的。例如，如果我们使用相对布局（relative layout）在屏幕底部放置一个按钮，用 `"adjustResize"` 来重新调整大小，使得按钮栏出现在输入法上方。

支持键盘导航

> 编写:zhaochunqi - 原文:http://developer.android.com/training/keyboard-input/navigation.html

除了软键盘输入法（如虚拟键盘）以外，Android支持将物理键盘连接到设备上。键盘不仅方便输入文本，而且提供一种方法来导航和与应用交互。尽管多数的手持设备（如手机）使用触摸作为主要的交互方式，但是随着平板和一些类似的设备正在逐步流行起来，许多用户开始喜欢外接键盘。

随着更多的Android设备提供这种体验，优化应用以支持通过键盘与应用进行交互变得越来越重要。这节课介绍了怎样为键盘导航提供更好的支持。

> **Note:** 对那些没有使用可见导航提示的应用来说，在应用中支持方向性的导航对于应用的可用性也是很重要的。在我们的应用中完全支持方向导航还可以帮助我们使用诸如uiautomator 等工具进行自动化用户界面测试。

测试应用

因为Android系统默认开启了大多必要的行为，所以用户可能已经可以在我们的应用中使用键盘导航了。

所有由Android framework（如Button和EditText）提供的交互部件是可获得焦点的。这意味着用户可以使用如D-pad或键盘等控制设备，并且当某个部件被选中时，部件会发光或者改变外观。

为了测试我们的应用：

1. 将应用安装到一个带有实体键盘的设备上。

   如果我们没有带实体键盘的设备，连接一个蓝牙键盘或者USB键盘(尽管并不是所有的设备都支持USB连接)

   我们还可以使用Android模拟器：

   i. 在AVD管理器中，要么点击**New Device**，要么选择一个已存在的文档点击**Clone**。

   ii. 在出现的窗口中，确保**Keyboard**和**D-pad**开启。

2. 为了验证我们的应用，只是用Tab键来进行UI导航，确保每一个UI控制的焦点与预期的一致。

   找到任何不在预期焦点的实例。

3. 从头开始，使用方向键(键盘上的箭头键)来控制应用的导航。

   在 UI 中每一个被选中的元素上，按上、下、左、右。

   找到每个不在预期焦点的实例。

如果我们找到任何使用Tab键或方向键后导航的效果不如预期的实例，那么在布局中指定焦点应该聚焦在哪里，如下面几部分所讨论的。

处理Tab导航

当用户使用键盘上的Tab键导航我们的应用时，系统会根据组件在布局中的显示顺序，在组件之间传递焦点。如果我们使用相对布局（relative layout），例如，在屏幕上的组件顺序与布局文件中组件的顺序不一致，那么我们可能需要手动指定焦点顺序。

举例来说，在下面的布局文件中，两个对齐右边的按钮和一个对齐第二个按钮左边的文本框。为了把焦点从第一个按钮传递到文本框，然后再传递到第二个按钮，布局文件需要使用属性 android:nextFocusForward，清楚地为每一个可被选中的组件定义焦点顺序：

```
<RelativeLayout ...>
    <Button
        android:id="@+id/button1"
        android:layout_alignParentTop="true"
        android:layout_alignParentRight="true"
        android:nextFocusForward="@+id/editText1"
        ... />
    <Button
        android:id="@+id/button2"
        android:layout_below="@id/button1"
        android:nextFocusForward="@+id/button1"
        ... />
    <EditText
        android:id="@id/editText1"
        android:layout_alignBottom="@+id/button2"
        android:layout_toLeftOf="@id/button2"
        android:nextFocusForward="@+id/button2"
        ...  />
    ...
</RelativeLayout>
```

现在焦点从 `button1` 到 `button2` 再到 `editText1` ，改成了按照在屏幕上出现的顺序：从 `button1` 到 `editText1` 再到 `button2` 。

处理方向导航

用户也能够使用键盘上的方向键在我们的app中导航(这种行为与在D-pad和轨迹球中的导航一致)。系统提供了一个最佳猜测：根据屏幕上 view 的布局，在给定的方向上，应该将交掉放在哪个 view 上。然而有时，系统会猜测错误。

当在给定的方向进行导航时，如果系统没有传递焦点给合适的 View，那么指定接收焦点的 view 来使用如下的属性：

- android:nextFocusUp
- android:nextFocusDown
- android:nextFocusLeft
- android:nextFocusRight

当用户导航到那个方向时，每一个属性指定了下一个接收焦点的 view，如根据 view ID 来指定。举例来说：

```
<Button
    android:id="@+id/button1"
    android:nextFocusRight="@+id/button2"
    android:nextFocusDown="@+id/editText1"
    ... />
<Button
    android:id="@id/button2"
    android:nextFocusLeft="@id/button1"
    android:nextFocusDown="@id/editText1"
    ... />
<EditText
    android:id="@id/editText1"
    android:nextFocusUp="@id/button1"
    ...  />
```

处理按键动作

> 编写:zhaochunqi - 原文:http://developer.android.com/training/keyboard-input/commands.html

当用户选中一个可编辑的文本 view（如 EditText 组件），而且用户连接了一个实体键盘时，所有输入由系统处理。然而，如果我们想接管或直接处理键盘输入，那么可以通过实现 KeyEvent.Callback 接口的回调方法，如 onKeyDown() 和 onKeyMultiple() 来完成上述目的。

因为 Activity 和 View 类都实现了 KeyEvent.Callback 接口，所以通常我们应该在这些类的继承中重写回调方法。

> **Note:** 当使用 KeyEvent 类和相关的 API 处理键盘事件时，我们应该期望这种键盘事件只从实体键盘发出。我们永远不应该依赖从一个软输入法（如屏幕键盘）来接收按键事件。

处理单个按键事件

处理单个的按键点击，需要适当地实现 onKeyDown() 或 onKeyUp()。通常，我们使用 onKeyUp() 来确保我们只接收一个事件。如果用户点击并按住按钮不放，onKeyDown() 会被调用多次。

举例，这个实现响应一些键盘按键来控制游戏：

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_D:
            moveShip(MOVE_LEFT);
            return true;
        case KeyEvent.KEYCODE_F:
            moveShip(MOVE_RIGHT);
            return true;
        case KeyEvent.KEYCODE_J:
            fireMachineGun();
            return true;
        case KeyEvent.KEYCODE_K:
            fireMissile();
            return true;
        default:
            return super.onKeyUp(keyCode, event);
    }
}
```

处理修饰键

为了对修饰键（例如将一个按键与 Shift 或者 Control 键组合）进行回应，我们可以查询 KeyEvent 来传递到回调方法。一些方法，如 getModifiers() 和 getMetaState()，提供一些关于修饰键的信息。然而，最简单的解决方案是用像 isShiftPressed() 和 isCtrlPressed() 等方法，检查我们关心的修饰键是否正在被按下。

例如，有一个 onKeyDown() 的实现，当Shift键和一个其他按键按下时，做一些额外的处理:

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        ...
        case KeyEvent.KEYCODE_J:
            if (event.isShiftPressed()) {
                fireLaser();
            } else {
                fireMachineGun();
            }
            return true;
        case KeyEvent.KEYCODE_K:
            if (event.isShiftPressed()) {
                fireSeekingMissle();
            } else {
                fireMissile();
            }
            return true;
        default:
            return super.onKeyUp(keyCode, event);
    }
}
```

支持游戏控制器

> 编写:heray1990 - 原文:http://developer.android.com/training/game-controllers/index.html

我们可以通过支持游戏控制器来增强用户体验。Android framework 提供了 APIs 来检测和处理游戏控制器的用户输入。

这节课介绍了如何使我们的游戏在不同的 Android API levels(API level 9或者更高) 间稳定地工作。还介绍了如何通过在我们的 App 中支持多个游戏控制器来增强用户体验。

Lessons

处理控制器输入动作

学习如何处理游戏控制器常用的输入元素，包括方向键按钮（D-pad）、游戏键盘和摇杆。

在不同的 **Android** 系统版本支持控制器

学习如何使游戏控制器在运行不同 Android 系统版本的设备上保持行为一致。

支持多个游戏控制器

学习如何检测和使用多个同时连接的游戏控制器。

处理控制器输入动作

> 编写:heray1990 - 原文:http://developer.android.com/training/game-controllers/controller-input.html

在系统层面上，Android 会以 Android 按键码值和坐标值的形式来报告来自游戏控制器的输入事件。在我们的游戏应用里，我们可以接收这些码值和坐标值，并将它们转化成特定的游戏行为。

当玩家将一个游戏控制器通过有线连接或者无线配对到 Android 设备时，系统会自动检测控制器，将它设置成输入设备并且开始报告它的输入事件。我们的游戏应用可以通过在活动的 Activity 或者被选中的 View 里调用下面这些回调方法，来接收上述输入事件（要么在 Activity，要么在 View 中实现实现这些回调方法，不要两个地方都实现回调）。

- 在 Activity 中：
  - dispatchGenericMotionEvent(android.view.MotionEvent)
    - 处理一般的运动事件，如摇动摇杆
  - dispatchKeyEvent(android.view.KeyEvent)
    - 处理按键事件，如按下或者释放游戏键盘的按键或者 D-pad 按钮。
- 在 View 中：
  - onGenericMotionEvent(android.view.MotionEvent)
    - 处理一般的运动事件，如摇动摇杆
  - onKeyDown(int, android.view.KeyEvent)
    - 处理按下一个按键的事件，如按下游戏键盘的按键或者 D-pad 按钮。
  - onKeyUp(int, android.view.KeyEvent)
    - 处理释放一个按键的事件，如释放游戏键盘的按键或者 D-pad 按钮。

建议的方法是从与用户交互的 View 对象捕获事件。请查看下面回调函数的对象，来获取关于接收到输入事件的类型：

KeyEvent：描述方向按键（D-pad）和游戏按键事件的对象。按键事件伴随着一个表示特定按键触发的按键码值*(key code)*，如 DPAD_DOWN 或者 BUTTON_A。我们可以通过调用 getKeyCode() 或者从按键事件回调方法（如 onKeyDown()）来获得按键码值。

MotionEvent：描述摇杆和肩键运动的输入。动作事件伴随着一个动作码（action code）和一系列坐标值（*axis values*）。动作码表示出现变化的状态，例如摇动一个摇杆。坐标值描述了特定物理操控的位置和其它运动属性，例如 AXIS_X 或者 AXIS_RTRIGGER。我们可以通过调用 getAction() 来获得动作码，通过调用 getAxisValue() 来获得坐标值。

这节课主要介绍如何通过实现上述的 View 回调方法与处理 KeyEvent 和 MotionEvent 对象，来处理常用控制器（游戏键盘按键、方向按键和摇杆）的输入。

验证游戏控制器是否已连接

在报告输入事件的时候，Android 不会区分游戏控制器事件与非游戏控制器事件。例如，一个触屏动作会产生一个表示触摸表面上 X 坐标的 AXIS_X，但是一个摇杆动作产生的 AXIS_X 则表示摇杆水平移动的位置。如果我们的游戏关注游戏控制器的输入，那么我们应该首先检测相应的事件来源类型。

通过调用 getSources() 来获得设备上支持的输入类型的位字段，来判断一个已连接的输入设备是不是一个游戏控制器。我们可以通过测试以查看下面的字段是否被设置：

- SOURCE_GAMEPAD 源类型表示输入设备有游戏手柄按键（如，BUTTON_A）。注意虽然一般的游戏手柄都会有方向控制键，但是这个源类型并不代表游戏控制器具有 D-pad 按钮。
- SOURCE_DPAD 源类型表示输入设备有 D-pad 按钮（如，DPAD_UP）。
- SOURCE_JOYSTICK 源类型表示输入设备有遥控杆（如，会通过 AXIS_X 和 AXIS_Y 记录动作的摇杆）。

下面的一小段代码介绍了一个 helper 方法，它的作用是让我们检验已接入的输入设备是否是游戏控制器。如果检测到是游戏控制器，那么这个方法会获得游戏控制器的设备 ID。然后，我们应该将每个设备 ID 与游戏中的玩家关联起来，并且单独处理每个已接入的玩家的游戏操作。想更详细地了解关于在一台Android设备中同时支持多个游戏控制器的方法，请见支持多个游戏控制器。

```
public ArrayList getGameControllerIds() {
    ArrayList gameControllerDeviceIds = new ArrayList();
    int[] deviceIds = InputDevice.getDeviceIds();
    for (int deviceId : deviceIds) {
        InputDevice dev = InputDevice.getDevice(deviceId);
        int sources = dev.getSources();

        // Verify that the device has gamepad buttons, control sticks, or both.
        if (((sources & InputDevice.SOURCE_GAMEPAD) == InputDevice.SOURCE_GAMEPAD)
                || ((sources & InputDevice.SOURCE_JOYSTICK)
                == InputDevice.SOURCE_JOYSTICK)) {
            // This device is a game controller. Store its device ID.
            if (!gameControllerDeviceIds.contains(deviceId)) {
                gameControllerDeviceIds.add(deviceId);
            }
        }
    }
    return gameControllerDeviceIds;
}
```

另外，我们可能想去检查已接入的单个游戏控制器的输入性能。这种检查在某些场合会很有用，例如，我们希望游戏只用到兼容的物理操控。

用下面这些方法检测一个游戏控制器是否支持一个特定的按键码或者坐标码：

- 在Android 4.4（API level 19）或者更高的系统中，调用 hasKeys(int) 来确定游戏控制器是否支持某个按键码。
- 在Android 3.1（API level 12）或者更高的系统中，首先调用 getMotionRanges()，然后在每个返回的 InputDevice.MotionRange 对象中调用 getAxis() 来获得坐标 ID。这样就可以得到游戏控制器支持的所有可用坐标轴。

处理游戏手柄按键

Figure 1介绍了 Android 如何将按键码和坐标值映射到实际的游戏手柄上。

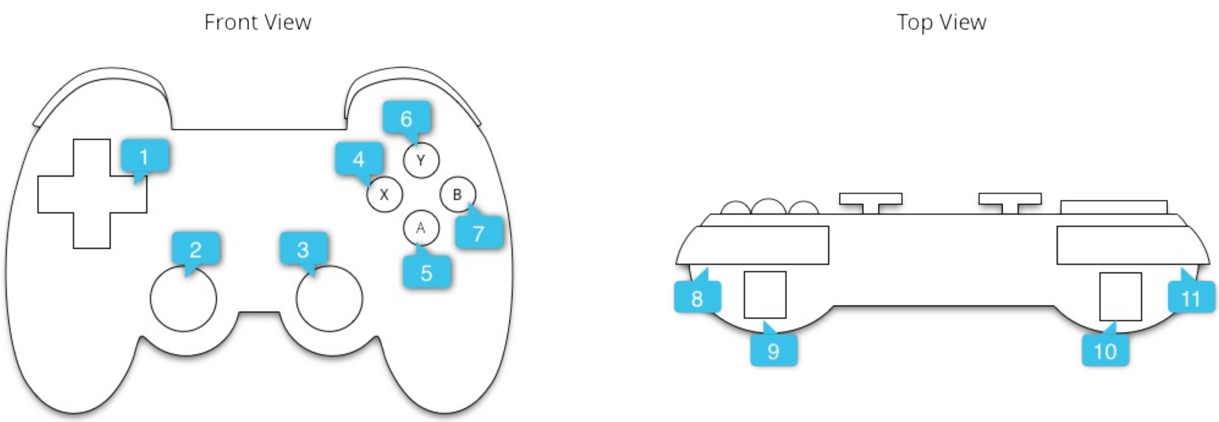


Figure 1. 一个常用的游戏手柄的外形

上图的标注对应下面的内容：

1. AXIS_HAT_X, AXIS_HAT_Y, DPAD_UP, DPAD_DOWN, DPAD_LEFT, DPAD_RIGHT
2. AXIS_X, AXIS_Y, BUTTON_THUMBL
3. AXIS_Z, AXIS_RZ, BUTTON_THUMBR
4. BUTTON_X
5. BUTTON_A
6. BUTTON_Y
7. BUTTON_B
8. BUTTON_R1
9. AXIS_RTRIGGER, AXIS_THROTTLE
10. AXIS_LTRIGGER, AXIS_BRAKE
11. BUTTON_L1

游戏手柄产生的通用的按键码包括 BUTTON_A、BUTTON_B、BUTTON_SELECT 和 BUTTON_START。当按下 D-pad 中间的交叉按键时，一些游戏控制器会触发 DPAD_CENTER 按键码。我们的游戏可以通过调用 getKeyCode() 或者从按键事件回调（如 onKeyDown()）得到按键码。如果一个事件与我们的游戏相关，那么将其处理成一个游戏动作。Table 1列出供大多数通用游戏手柄按钮使用的推荐游戏动作。

Table 1. 供游戏手柄使用的推荐游戏动作

游戏动作	按键码
在主菜单中启动游戏，或者在游戏过程中暂停/取消暂停	BUTTON_START
显示菜单	BUTTON_SELECT 和 KEYCODE_MENU
跟Android导航设计指导中的Back导航行为一样	KEYCODE_BACK
返回到菜单中上一项	BUTTON_B
确认选择，或者执行主要的游戏动作	BUTTON_A 和 DPAD_CENTER

* 我们的游戏不应该依赖于*Start*、*Select*或者*Menu*按键的存在。

> **Tip:** 可以考虑在游戏中提供一个配置界面，使得用户可以个性化游戏控制器与游戏动作的映射。

下面的代码介绍了如何重写 onKeyDown() 来将 BUTTON_A 和 DPAD_CENTER 按钮结合到一个游戏动作。

```
public class GameView extends View {
    ...

    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        boolean handled = false;
        if ((event.getSource() & InputDevice.SOURCE_GAMEPAD)
                == InputDevice.SOURCE_GAMEPAD) {
            if (event.getRepeatCount() == 0) {
                switch (keyCode) {
                    // Handle gamepad and D-pad button presses to
                    // navigate the ship
                    ...

                    default:
                         if (isFireKey(keyCode)) {
                             // Update the ship object to fire lasers
                             ...
                             handled = true;
                         }
                     break;
                }
            }
            if (handled) {
                return true;
            }
        }
        return super.onKeyDown(keyCode, event);
    }

    private static boolean isFireKey(int keyCode) {
        // Here we treat Button_A and DPAD_CENTER as the primary action
        // keys for the game.
        return keyCode == KeyEvent.KEYCODE_DPAD_CENTER
                || keyCode == KeyEvent.KEYCODE_BUTTON_A;
    }
}
```

> **Note:** 在 Android 4.2（API level 17）和更低版本的系统中，系统默认会把 BUTTON_A 当作 Android *Back*（返回）键。如果我们的应用支持这些 Android 版本，请确保将 BUTTON_A 转换成主要的游戏动作。引用 Build.VERSION.SDK_INT 值来决定设备上当前的 Android SDK 版本。

处理 D-pad 输入

四方向的方向键（D-pad）在很多游戏控制器中是一种很常见的物理控制。Android 将 D-pad 的上和下按键按压报告成 AXIS_HAT_Y 事件（范围从-1.0（上）到1.0（下）），将 D-pad 的左或者右按键按压报告成 AXIS_HAT_X 事件（范围从-1.0（左）到1.0（右））。

一些游戏控制器会将 D-pad 按压报告成一个按键码。如果我们的游戏有检测 D-pad 的按压，那么我们应该将坐标值事件和 D-pad 按键码当成一样的输入事件，如 table 2 介绍的一样。

Table 2. D-pad 按键码和坐标值的推荐默认游戏动作。

游戏动作	D-pad 按键码	坐标值
向上	KEYCODE_DPAD_UP	AXIS_HAT_Y（从 0 到 -1.0）
向下	KEYCODE_DPAD_DOWN	AXIS_HAT_Y（从 0 到 1.0）
向左	KEYCODE_DPAD_LEFT	AXIS_HAT_X（从 0 到 -1.0）
向右	KEYCODE_DPAD_RIGHT	AXIS_HAT_X（从 0 到 1.0）

下面的代码介绍了通过一个 helper 类，来检查从一个输入事件来决定 D-pad 方向的坐标值和按键码。

```
public class Dpad {
    final static int UP       = 0;
    final static int LEFT     = 1;
    final static int RIGHT    = 2;
    final static int DOWN     = 3;
    final static int CENTER   = 4;

    int directionPressed = -1; // initialized to -1

    public int getDirectionPressed(InputEvent event) {
        if (!isDpadDevice(event)) {
           return -1;
        }

        // If the input event is a MotionEvent, check its hat axis values.
        if (event instanceof MotionEvent) {

            // Use the hat axis value to find the D-pad direction
            MotionEvent motionEvent = (MotionEvent) event;
            float xaxis = motionEvent.getAxisValue(MotionEvent.AXIS_HAT_X);
            float yaxis = motionEvent.getAxisValue(MotionEvent.AXIS_HAT_Y);

            // Check if the AXIS_HAT_X value is -1 or 1, and set the D-pad
            // LEFT and RIGHT direction accordingly.
            if (Float.compare(xaxis, -1.0f) == 0) {
                directionPressed =  Dpad.LEFT;
            } else if (Float.compare(xaxis, 1.0f) == 0) {
                directionPressed =  Dpad.RIGHT;
            }
            // Check if the AXIS_HAT_Y value is -1 or 1, and set the D-pad
```

```
            // UP and DOWN direction accordingly.
            else if (Float.compare(yaxis, -1.0f) == 0) {
                directionPressed =  Dpad.UP;
            } else if (Float.compare(yaxis, 1.0f) == 0) {
                directionPressed =  Dpad.DOWN;
            }
        }

        // If the input event is a KeyEvent, check its key code.
        else if (event instanceof KeyEvent) {

           // Use the key code to find the D-pad direction.
            KeyEvent keyEvent = (KeyEvent) event;
            if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_LEFT) {
                directionPressed = Dpad.LEFT;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_RIGHT) {
                directionPressed = Dpad.RIGHT;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_UP) {
                directionPressed = Dpad.UP;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_DOWN) {
                directionPressed = Dpad.DOWN;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_CENTER) {
                directionPressed = Dpad.CENTER;
            }
        }
        return directionPressed;
    }

    public static boolean isDpadDevice(InputEvent event) {
        // Check that input comes from a device with directional pads.
        if ((event.getSource() & InputDevice.SOURCE_DPAD)
             != InputDevice.SOURCE_DPAD) {
             return true;
         } else {
             return false;
         }
     }
}
```

我们可以在任意想要处理 D-pad 输入（例如，在 onGenericMotionEvent() 或者 onKeyDown() 回调函数）的地方使用这个 helper 类。

例如：

```
Dpad mDpad = new Dpad();
...
@Override
public boolean onGenericMotionEvent(MotionEvent event) {

    // Check if this event if from a D-pad and process accordingly.
    if (Dpad.isDpadDevice(event)) {

       int press = mDpad.getDirectionPressed(event);
       switch (press) {
            case LEFT:
                // Do something for LEFT direction press
                ...
                return true;
            case RIGHT:
                // Do something for RIGHT direction press
                ...
                return true;
            case UP:
                // Do something for UP direction press
                ...
                return true;
            ...
        }
    }

    // Check if this event is from a joystick movement and process accordingly.
    ...
}
```

处理摇杆动作

当玩家移动游戏控制器上的摇杆时，Android 会报告一个包含 ACTION_MOVE 动作码和更新摇杆在坐标轴的位置的 MotionEvent。我们的游戏可以使用 MotionEvent 提供的数据来确定是否发生摇杆的动作。

注意到摇杆移动会在单个对象中批处理多个移动示例。MotionEvent 对象包含每个摇杆坐标当前的位置和每个坐标轴上的多个历史位置。当用 ACTION_MOVE 动作码（例如摇杆移动）来报告移动事件时，Android 会高效地批处理坐标值。由坐标值组成的不同的历史值比当前的坐标值要旧，比之前报告的任意移动事件要新。详情见 MotionEvent 参考文档。

我们可以使用历史信息，根据摇杆输入更精确地表达游戏对象的活动。调用 getAxisValue() 或者 getHistoricalAxisValue() 来获取现在和历史的值。我们也可以通过调用 getHistorySize() 来找到摇杆事件的历史点号码。

下面的代码介绍了如何重写 onGenericMotionEvent() 回调函数来处理摇杆输入。我们应该首先处理历史坐标值，然后处理当前值。

```java
public class GameView extends View {

    @Override
    public boolean onGenericMotionEvent(MotionEvent event) {

        // Check that the event came from a game controller
        if ((event.getSource() & InputDevice.SOURCE_JOYSTICK) ==
                InputDevice.SOURCE_JOYSTICK &&
                event.getAction() == MotionEvent.ACTION_MOVE) {

            // Process all historical movement samples in the batch
            final int historySize = event.getHistorySize();

            // Process the movements starting from the
            // earliest historical position in the batch
            for (int i = 0; i < historySize; i++) {
                // Process the event at historical position i
                processJoystickInput(event, i);
            }

            // Process the current movement sample in the batch (position -1)
            processJoystickInput(event, -1);
            return true;
        }
        return super.onGenericMotionEvent(event);
    }
}
```

在使用摇杆输入之前，我们需要确定摇杆是否居中，然后计算相应的坐标移动距离。一般摇杆会有一个平面区，即在坐标 (0, 0) 附近一个值范围内的坐标点都被当作是中点。如果 Android 系统报告坐标值掉落在平面区内，那么我们应该认为控制器处于静止（即沿着 x、y 两个坐标轴都是静止的）。

下面的代码介绍了一个用于计算沿着每个坐标轴的移动距离的 helper 方法。我们将在后面讨论的 `processJoystickInput()` 方法中调用这个 helper 方法。

```
private static float getCenteredAxis(MotionEvent event,
        InputDevice device, int axis, int historyPos) {
    final InputDevice.MotionRange range =
            device.getMotionRange(axis, event.getSource());

    // A joystick at rest does not always report an absolute position of
    // (0,0). Use the getFlat() method to determine the range of values
    // bounding the joystick axis center.
    if (range != null) {
        final float flat = range.getFlat();
        final float value =
                historyPos < 0 ? event.getAxisValue(axis):
                event.getHistoricalAxisValue(axis, historyPos);

        // Ignore axis values that are within the 'flat' region of the
        // joystick axis center.
        if (Math.abs(value) > flat) {
            return value;
        }
    }
    return 0;
}
```

将它们都放在一起，下面是我们如何在游戏中处理摇杆移动：

```
private void processJoystickInput(MotionEvent event,
        int historyPos) {

    InputDevice mInputDevice = event.getDevice();

    // Calculate the horizontal distance to move by
    // using the input value from one of these physical controls:
    // the left control stick, hat axis, or the right control stick.
    float x = getCenteredAxis(event, mInputDevice,
            MotionEvent.AXIS_X, historyPos);
    if (x == 0) {
        x = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_HAT_X, historyPos);
    }
    if (x == 0) {
        x = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_Z, historyPos);
    }

    // Calculate the vertical distance to move by
    // using the input value from one of these physical controls:
    // the left control stick, hat switch, or the right control stick.
    float y = getCenteredAxis(event, mInputDevice,
            MotionEvent.AXIS_Y, historyPos);
    if (y == 0) {
        y = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_HAT_Y, historyPos);
    }
    if (y == 0) {
        y = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_RZ, historyPos);
    }

    // Update the ship object based on the new x and y values
}
```

为了支持除了单个摇杆之外更多复杂的功能，按照下面的做法：

- 处理两个控制器摇杆。很多游戏控制器左右两边都有摇杆。对于左摇杆，Android 会报告水平方向的移动为 AXIS_X 事件，垂直方向的移动为 AXIS_Y 事件。对于右摇杆，Android 会报告水平方向的移动为 AXIS_Z 事件，垂直方向的移动为 AXIS_RZ 事件。确保在代码中处理两个摇杆。
- 处理肩键按压（但需要提供另一种输入方法）。一些控制器会有左右肩键。如果存在这些按键，那么 Android 报告左肩键按压为一个 AXIS_LTRIGGER 事件，右肩键按压为一个 AXIS_RTRIGGER 事件。在 Android 4.3（API level 18）中，一个产生了 AXIS_LTRIGGER 事件的控制器也会报告一个完全一样的 AXIS_BRAKE 坐标值。同

样，AXIS_RTRIGGER 对应 AXIS_GAS。Android 会报告模拟按键按压为从 0.0（释放）到 1.0（按下）的标准值。并不是所有的控制器都有肩键，所以需要允许玩家用其它按钮来执行那些游戏动作。

在不同的 Android 系统版本支持控制器

> 编写:heray1990 - 原文:http://developer.android.com/training/game-controllers/compatibility.html

如果我们正为游戏提供游戏控制器的支持，那么我们需要确保我们的游戏对于运行着不同 Android 版本的设备对控制器都有一致的响应。这会使得我们的游戏扩大用户群体，同时，我们的玩家可以享受即使他们切换或者升级 Android 设备的时候，都可以使用他们的控制器无缝对接的游戏体验。

这节课展示了如何用向下兼容的方式使用 Android 4.1 或者更高版本中可用的 API，使我们的游戏运行在 Android 2.3 或者更高的设备上时，支持下面的功能：

- 游戏可以检测是否有一个新的游戏控制器接入、变更或者移除。
- 游戏可以查询游戏控制器的兼容性。
- 游戏可以识别从游戏控制器传入的动作事件。

这节课的例子是基于 `ControllerSample.zip` 提供的参考实现。这个示例介绍了如何实现 `InputManagerCompat` 接口来支持不同的 Android 版本。我们必须使用 Android 4.1（API level 16）或者更高的版本来编译这个示例代码。一旦编译完成，生成的示例 app 可以在任何运行着 Android 2.3（API level 9）或者更高版本的设备上运行。

准备支持游戏控制器的抽象 API

假设我们想确定在运行着 Android 2.3（API level 9）的设备上，游戏控制器的连接状态是否发生改变。无论如何，API 只在 Android 4.1（API level 16）或者更高的版本上可用，所以我们需要提供一个支持 Android 4.1（API level 16）或者更高版本的实现方法的同时，提供一个支持从 Android 2.3 到 Android 4.0 的回退机制。

为了帮助我们确定哪个功能需要这样的回退机制，table 1 列出了 Android 2.3（API level 9）、3.1（API level 12）和 4.1（API level 16）之间，对于支持游戏控制器的不同之处。

Table 1. API 在不同 Android 版本间对游戏控制器支持的不同点

Controller Information	Controller API	API level 9	API level 12	API level 16
Device Identification	getInputDeviceIds()			*
	getInputDevice()			*
	getVibrator()			*
	SOURCE_JOYSTICK		*	*
	SOURCE_GAMEPAD		*	*
Connection Status	onInputDeviceAdded()			*
	onInputDeviceChanged()			*
	onInputDeviceRemoved()			*
Input Event Identification	D-pad press (KEYCODE_DPAD_UP, KEYCODE_DPAD_DOWN, KEYCODE_DPAD_LEFT, KEYCODE_DPAD_RIGHT, KEYCODE_DPAD_CENTER)	*	*	*
	Gamepad button press (BUTTON_A, BUTTON_B, BUTTON_THUMBL, BUTTON_THUMBR, BUTTON_SELECT, BUTTON_START, BUTTON_R1, BUTTON_L1, BUTTON_R2, BUTTON_L2)		*	*
	Joystick and hat switch movement (AXIS_X, AXIS_Y, AXIS_Z, AXIS_RZ, AXIS_HAT_X, AXIS_HAT_Y)		*	*
	Analog trigger press (AXIS_LTRIGGER, AXIS_RTRIGGER)		*	*

我们可以使用抽象化概念来建立能够工作在不同平台的版本识别的游戏控制器支持。这种方法包括下面几个步骤：

1. 定义一个中间 Java 接口来抽象化我们游戏需要的游戏控制器功能的实现。
2. 创建一个使用 Android 4.1 和更高版本 API 的接口的代理实现。
3. 创建一个使用 Android 2.3 到 Android 4.0 之间可用的 API 的接口的自定义实现。
4. 创建在运行时，在这上述这些实现之间切换的逻辑，并且开始使用我们游戏中的接口。

有关如何使用抽象化概念来保证应用可以在不同版本的 Android 之间，以向后兼容的方式工作的概述，请见创建向后兼容的 UI。

添加向后兼容的接口

对于向后兼容，我们可以创建一个自定义接口，然后添加特定版本的实现。这种方法的一个优点是它可以让我们借鉴 Android 4.1（API level 16）上支持游戏控制器的公共接口。

```
// The InputManagerCompat interface is a reference example.
// The full code is provided in the ControllerSample.zip sample.
public interface InputManagerCompat {
    ...
    public InputDevice getInputDevice(int id);
    public int[] getInputDeviceIds();

    public void registerInputDeviceListener(
            InputManagerCompat.InputDeviceListener listener,
            Handler handler);
    public void unregisterInputDeviceListener(
            InputManagerCompat.InputDeviceListener listener);

    public void onGenericMotionEvent(MotionEvent event);

    public void onPause();
    public void onResume();

    public interface InputDeviceListener {
        void onInputDeviceAdded(int deviceId);
        void onInputDeviceChanged(int deviceId);
        void onInputDeviceRemoved(int deviceId);
    }
    ...
}
```

`InputManagerCompat` 接口提供了下面的方法：

`getInputDevice()`

借鉴 getInputDevice()。包括代表一个游戏控制器兼容性的 InputDevice 对象。

`getInputDeviceIds()`

借鉴getInputDeviceIds()。返回一个整型数组，每一个数组成员表示一个不同输入设备的 ID。这对于想要构建一个支持多玩家和检测连接了多少个控制器的游戏是很有用的。

`registerInputDeviceListener()`

借鉴registerInputDeviceListener()。注册一个监听器，当一个新的设备添加、改变或者移除的时候，我们会收到通知。

`unregisterInputDeviceListener()`

借鉴unregisterInputDeviceListener()。注销一个输入设备监听器。

`onGenericMotionEvent()`

借鉴onGenericMotionEvent()。让我们的游戏截取和处理 MotionEvent 对象和代表类似移动摇杆和按下模拟触发器等事件的坐标值。

`onPause()`

当主 activity 暂停或者当游戏不再聚焦时，停止轮询游戏控制器事件。

`onResume()`

当主 activity 恢复或者当游戏开始和在前台运行时，启动轮询游戏控制器事件。

`InputDeviceListener`

借鉴 InputManager.InputDeviceListener 接口。当添加、改变或者移除游戏控制器时，会通知我们的游戏。

下一步，创建 `InputManagerCompat` 的实现，使得可以在不同平台版本间工作。如果我们的游戏运行在 Android 4.1 或者更高版本，调用 `InputManagerCompat` 方法，代理实现调用在 InputManager 中等效的方法。然而，如果我们的游戏运行在 Andoird 2.3 到 Android 4.0，自定义的实现过程通过使用不晚于 Android 2.3 引进的 API 来调用 `InputManagerCompat` 方法。不管在运行时使用哪种特定版本的实现，实现会透明地将回调结果传给游戏。

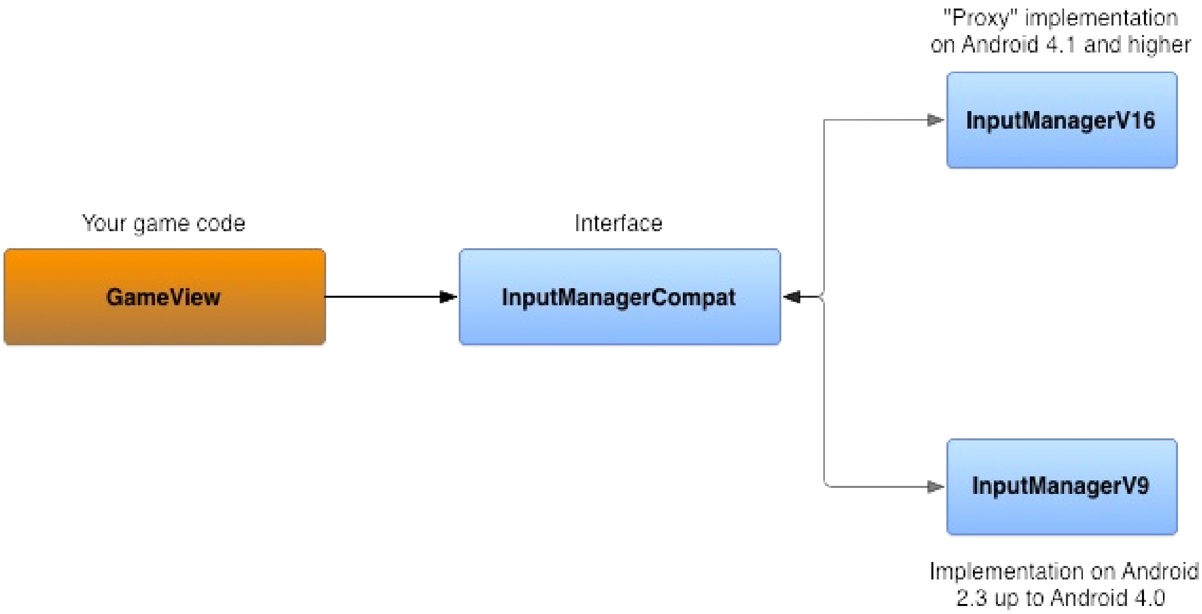


Figure 1. 接口和特定版本实现的类图。

实现 Android 4.1 和更高版本的接口

`InputManagerCompatV16` 是 `InputManagerCompat` 接口的实现，该接口代理方法调用一个 InputManager 和 InputManager.InputDeviceListener。InputManager是从系统 Context 得到。

```java
// The InputManagerCompatV16 class is a reference implementation.
// The full code is provided in the ControllerSample.zip sample.
public class InputManagerV16 implements InputManagerCompat {

    private final InputManager mInputManager;
    private final Map mListeners;

    public InputManagerV16(Context context) {
        mInputManager = (InputManager)
                context.getSystemService(Context.INPUT_SERVICE);
        mListeners = new HashMap();
    }

    @Override
    public InputDevice getInputDevice(int id) {
        return mInputManager.getInputDevice(id);
    }

    @Override
    public int[] getInputDeviceIds() {
        return mInputManager.getInputDeviceIds();
    }

    static class V16InputDeviceListener implements
            InputManager.InputDeviceListener {
        final InputManagerCompat.InputDeviceListener mIDL;

        public V16InputDeviceListener(InputDeviceListener idl) {
            mIDL = idl;
        }

        @Override
        public void onInputDeviceAdded(int deviceId) {
            mIDL.onInputDeviceAdded(deviceId);
        }

        // Do the same for device change and removal
        ...
    }

    @Override
    public void registerInputDeviceListener(InputDeviceListener listener,
            Handler handler) {
        V16InputDeviceListener v16Listener = new
                V16InputDeviceListener(listener);
        mInputManager.registerInputDeviceListener(v16Listener, handler);
        mListeners.put(listener, v16Listener);
    }

    // Do the same for unregistering an input device listener
    ...
```

```
    @Override
    public void onGenericMotionEvent(MotionEvent event) {
        // unused in V16
    }

    @Override
    public void onPause() {
        // unused in V16
    }

    @Override
    public void onResume() {
        // unused in V16
    }

}
```

实现 Android 2.3 到 Android 4.0 的接口

`InputManagerV9` 实现使用了不晚于 Android 2.3 引进的 API。为了创建一个支持 Android 2.3 到 Android 4.0 的 `InputManagerCompat` 实现，我们可以使用下面的对象：

- 设备 ID 的 `SparseArray` 跟踪已连接到设备的游戏控制器。
- 一个 `Handler` 来处理设备事件。当一个 app 启动或者恢复时，`Handler` 接收一个消息来开始轮询游戏控制器的断开。`Handler` 将启动一个循环来检查每个已知连接的游戏控制器并且查看是否返回一个设备 ID。返回 `null` 表示游戏控制器断开。当 app 暂停时，`Handler` 停止轮询。
- 一个 `InputManagerCompat.InputDeviceListener` 的 `Map` 对象。我们会使用这个 listener 来更新跟踪游戏遥控器的连接状态。

```
// The InputManagerCompatV9 class is a reference implementation.
// The full code is provided in the ControllerSample.zip sample.
public class InputManagerV9 implements InputManagerCompat {
    private final SparseArray mDevices;
    private final Map mListeners;
    private final Handler mDefaultHandler;
    …

    public InputManagerV9() {
        mDevices = new SparseArray();
        mListeners = new HashMap();
        mDefaultHandler = new PollingMessageHandler(this);
    }
}
```

实现继承 `Handler` 的 `PollingMessageHandler`，并重写 `handleMessage()` 方法。这个方法检查已连接的游戏控制器是否已经断开并且通知已注册的 listener。

```
private static class PollingMessageHandler extends Handler {
    private final WeakReference mInputManager;

    PollingMessageHandler(InputManagerV9 im) {
        mInputManager = new WeakReference(im);
    }

    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
            case MESSAGE_TEST_FOR_DISCONNECT:
                InputManagerV9 imv = mInputManager.get();
                if (null != imv) {
                    long time = SystemClock.elapsedRealtime();
                    int size = imv.mDevices.size();
                    for (int i = 0; i < size; i++) {
                        long[] lastContact = imv.mDevices.valueAt(i);
                        if (null != lastContact) {
                            if (time - lastContact[0] > CHECK_ELAPSED_TIME) {
                                // check to see if the device has been
                                // disconnected
                                int id = imv.mDevices.keyAt(i);
                                if (null == InputDevice.getDevice(id)) {
                                    // Notify the registered listeners
                                    // that the game controller is disconnected
                                    ...
                                    imv.mDevices.remove(id);
                                } else {
                                    lastContact[0] = time;
                                }
                            }
                        }
                    }
                    sendEmptyMessageDelayed(MESSAGE_TEST_FOR_DISCONNECT,
                            CHECK_ELAPSED_TIME);
                }
                break;
        }
    }
}
```

至于启动和停止轮询游戏控制器的断开，重写这些方法：

```
private static final int MESSAGE_TEST_FOR_DISCONNECT = 101;
private static final long CHECK_ELAPSED_TIME = 3000L;

@Override
public void onPause() {
    mDefaultHandler.removeMessages(MESSAGE_TEST_FOR_DISCONNECT);
}

@Override
public void onResume() {
    mDefaultHandler.sendEmptyMessageDelayed(MESSAGE_TEST_FOR_DISCONNECT,
            CHECK_ELAPSED_TIME);
}
```

重写 `onGenericMotionEvent()` 方法检测输入设备是否已添加。当系统通知一个动作事件时，检查这个事件是否从已经跟踪过的还是新的设备 ID 中发出。如果是新的设备 ID，通知已注册的 listener。

```
@Override
public void onGenericMotionEvent(MotionEvent event) {
    // detect new devices
    int id = event.getDeviceId();
    long[] timeArray = mDevices.get(id);
    if (null == timeArray) {
        // Notify the registered listeners that a game controller is added
        ...
        timeArray = new long[1];
        mDevices.put(id, timeArray);
    }
    long time = SystemClock.elapsedRealtime();
    timeArray[0] = time;
}
```

listener 的通知通过使用 `Handler` 对象发送一个 `DeviceEvent` `Runnable` 对象到消息队列来实现。`DeviceEvent` 包含了一个 `InputManagerCompat.InputDeviceListener` 的引用。当 `DeviceEvent` 运行时，适当的 listener 回调方法会被调用，标志游戏控制器是否被添加、改变或者移除。

```
@Override
public void registerInputDeviceListener(InputDeviceListener listener,
        Handler handler) {
    mListeners.remove(listener);
    if (handler == null) {
        handler = mDefaultHandler;
    }
    mListeners.put(listener, handler);
}
```

```
@Override
public void unregisterInputDeviceListener(InputDeviceListener listener) {
    mListeners.remove(listener);
}

private void notifyListeners(int why, int deviceId) {
    // the state of some device has changed
    if (!mListeners.isEmpty()) {
        for (InputDeviceListener listener : mListeners.keySet()) {
            Handler handler = mListeners.get(listener);
            DeviceEvent odc = DeviceEvent.getDeviceEvent(why, deviceId,
                    listener);
            handler.post(odc);
        }
    }
}

private static class DeviceEvent implements Runnable {
    private int mMessageType;
    private int mId;
    private InputDeviceListener mListener;
    private static Queue sObjectQueue =
            new ArrayDeque();
    ...

    static DeviceEvent getDeviceEvent(int messageType, int id,
            InputDeviceListener listener) {
        DeviceEvent curChanged = sObjectQueue.poll();
        if (null == curChanged) {
            curChanged = new DeviceEvent();
        }
        curChanged.mMessageType = messageType;
        curChanged.mId = id;
        curChanged.mListener = listener;
        return curChanged;
    }

    @Override
    public void run() {
        switch (mMessageType) {
            case ON_DEVICE_ADDED:
                mListener.onInputDeviceAdded(mId);
                break;
            case ON_DEVICE_CHANGED:
                mListener.onInputDeviceChanged(mId);
                break;
            case ON_DEVICE_REMOVED:
                mListener.onInputDeviceRemoved(mId);
                break;
            default:
                // Handle unknown message type
                ...
                break;
```

```
        }
        // Put this runnable back in the queue
        sObjectQueue.offer(this);
    }
}
```

我们现在已经有两个 `InputManagerCompat` 的实现：一个可以在运行 Android 4.1 或者更高版本的设备上工作，另一个可以在运行 Android 2.3 到 Android 4.0 的设备上工作。

使用特定版本的实现

特定版本切换的逻辑是在一个充当 factory 的类中实现。

```
public static class Factory {
    public static InputManagerCompat getInputManager(Context context) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
            return new InputManagerV16(context);
        } else {
            return new InputManagerV9();
        }
    }
}
```

现在我们可以简单地实例化一个 `InputManagerCompat` 对象，并且在主 `View` 中注册 `InputManagerCompat.InputDeviceListener`。由于我们建立的版本切换逻辑，我们的游戏会自动为设备上运行的 Android 版本使用适当的实现。

```
public class GameView extends View implements InputDeviceListener {
    private InputManagerCompat mInputManager;
    ...

    public GameView(Context context, AttributeSet attrs) {
        mInputManager =
                InputManagerCompat.Factory.getInputManager(this.getContext());
        mInputManager.registerInputDeviceListener(this, null);
        ...
    }
}
```

下一步，重写主 View 的 `onGenericMotionEvent()` 方法，详见处理从游戏控制器传来的 MotionEvent。我们的游戏现在应该可以一致地处理运行着 Android 2.3（API level 9）和更高版本设备上的游戏控制器事件。

```
@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    mInputManager.onGenericMotionEvent(event);

    // Handle analog input from the controller as normal
    ...
    return super.onGenericMotionEvent(event);
}
```

我们可以在上述的 `ControllerSample.zip` 示例的 `GameView` 类中找到这个兼容性的完整的代码。

支持多个游戏控制器

> 编写:heray1990 - 原文:http://developer.android.com/training/game-controllers/multiple-controllers.html

尽管大部分的游戏都被设计成一台 Android 设备支持一个用户，但是仍然有可能支持在同一台 Android 设备上同时连接的多个游戏控制器（即多个用户）。

这节课覆盖了一些处理单个设备多个玩家（或者多个控制器）输入的基本技术。这包括维护一个在玩家化身和每个控制器之间的映射，以及适当地处理控制器的输入事件。

映射玩家到控制器的设备 ID

当一个游戏控制器连接到一台 Android 设备，系统会为控制器指定一个整型的设备 ID。我们可以通过调用 `InputDevice.getDeviceIds()` 来取得已连接的游戏控制器的设备 ID，如验证游戏控制器是否已连接介绍的一样。我们可以将每个设备 ID 与游戏中的玩家关联起来，然后分别处理每个玩家的游戏动作。

> **Note**：在运行着 Android 4.1（API level 16）或者更高版本的设备上，我们可以通过使用 `getDescriptor()` 来取得输入设备的描述符。上述函数为输入设备返回一个唯一连续的字符串值。不同于设备 ID，即使在输入设备断开、重连或者重新配置时，描述符都不会变化。

下面的代码介绍了如何使用 SparseArray 来关联玩家化身与一个特定的控制器。在这个例子中，`mShips` 变量保存了一个 `Ship` 对象的集合。当一个新的控制器连接到一个用户时，会创建一个新的玩家化身。当已关联的控制器被移除时，对应的玩家化身会被移除。

`onInputDeviceAdded()` 和 `onInputDeviceRemoved()` 回调函数是在不同的 Android 系统版本支持控制器中介绍的抽象层的一部分。通过实现这些 listener 回调，我们的游戏可以在添加或者移除控制器的时候，识别出游戏控制器的设备 ID。这个检测兼容 Android 2.3（API level 9）和更高的版本。

```
private final SparseArray<Ship> mShips = new SparseArray<Ship>();

@Override
public void onInputDeviceAdded(int deviceId) {
    getShipForID(deviceId);
}

@Override
public void onInputDeviceRemoved(int deviceId) {
    removeShipForID(deviceId);
}

private Ship getShipForID(int shipID) {
    Ship currentShip = mShips.get(shipID);
    if ( null == currentShip ) {
        currentShip = new Ship();
        mShips.append(shipID, currentShip);
    }
    return currentShip;
}

private void removeShipForID(int shipID) {
    mShips.remove(shipID);
}
```

处理多个控制器输入

我们的游戏应该执行下面的循环来处理多个控制器的输入：

1. 检测是否出现一个输入事件。
2. 识别输入源和它的设备 ID。
3. 根据以输入事件按键码或者坐标值的形式表示的 action，更新玩家化身与设备 ID 的关联关系。
4. 渲染和更新用户界面。

keyEvent 和 MotionEvent 输入事件与设备 ID 相关联。我们的游戏可以利用这个关联来确定输入事件从哪个控制器发出，并且更新玩家化身与控制器的关联。

下面的代码介绍了我们如何将一个玩家化身引用相应的游戏控制器设备 ID，并且根据用户按下控制器的按键来更新游戏。

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if ((event.getSource() & InputDevice.SOURCE_GAMEPAD)
                == InputDevice.SOURCE_GAMEPAD) {
        int deviceId = event.getDeviceId();
        if (deviceId != -1) {
            Ship currentShip = getShipForId(deviceId);
            // Based on which key was pressed, update the player avatar
            // (e.g. set the ship headings or fire lasers)
            ...
            return true;
        }
    }
    return super.onKeyDown(keyCode, event);
}
```

> **Note**：一个最佳做法，当用户的游戏控制器断开时，我们应该停止游戏并询问用户是否像要重新连接。

Android后台任务

> 编写:kesenhoo - 原文:http://developer.android.com/training/best-background.html

下面的这些课程会教我们如何通过把任务执行在后台线程来提升程序的性能，还会教我们如何最小化后台线程对电量的消耗。

在后台Service中执行任务

学习如何通过发送任务给后台Service来提升UI的性能并避免ANR。

在后台加载数据

学习如何使用CursorLoader来查询数据，同时避免影响到UI的响应性。

管理设备的唤醒状态

学习如何使用重复闹钟以及唤醒锁来执行后台任务。

在IntentService中执行后台任务

> 编写:kesenhoo - 原文:http://developer.android.com/training/run-background-service/index.html

除非我们特别为某个操作指定特定的线程，否则大部分在前台UI界面上的操作任务都执行在一个叫做UI Thread的特殊线程中。这可能存在某些隐患，因为部分在UI界面上的耗时操作可能会影响界面的响应性能。UI界面的性能问题会容易惹恼用户，甚至可能导致系统ANR错误。为了避免这样的问题，Android Framework提供了几个类，用来帮助你把那些耗时操作移动到后台线程中执行。那些类中最常用的就是IntentService.

这一章节会讲到如何实现一个IntentService，向它发送任务并反馈任务的结果给其他模块。

Demos

ThreadSample.zip

Lessons

- 创建IntentService

  学习如何创建一个IntentService。

- 发送任务请求给IntentService

  学习如何发送工作任务给IntentService。

- 报告后台任务的执行状态

  学习如何使用Intent与LocalBroadcastManager在Activit与IntentService之间进行交互。

创建后台服务

> 编写:kesenhoo - 原文:http://developer.android.com/training/run-background-service/create-service.html

IntentService为在单一后台线程中执行任务提供了一种直接的实现方式。它可以处理一个耗时的任务并确保不影响到UI的响应性。另外IntentService的执行还不受UI生命周期的影响，以此来确保AsyncTask能够顺利运行。

但是IntentService有下面几个局限性：

- 不可以直接和UI做交互。为了把他执行的结果体现在UI上，需要把结果返回给Activity。
- 工作任务队列是顺序执行的，如果一个任务正在IntentService中执行，此时你再发送一个新的任务请求，这个新的任务会一直等待直到前面一个任务执行完毕才开始执行。
- 正在执行的任务无法打断。

虽然有上面那些限制，然而在在大多数情况下，IntentService都是执行简单后台任务操作的理想选择。

这节课会演示如何创建继承的IntentService。同样也会演示如何创建必须的回调方法 `onHandleIntent()` 。最后，还会解释如何在manifest文件中定义这个IntentService。

1)创建IntentService

为你的app创建一个IntentService组件，需要自定义一个新的类，它继承自IntentService，并重写onHandleIntent()方法，如下所示：

```
public class RSSPullService extends IntentService {
    @Override
    protected void onHandleIntent(Intent workIntent) {
        // Gets data from the incoming Intent
        String dataString = workIntent.getDataString();
        ...
        // Do work here, based on the contents of dataString
        ...
    }
}
```

注意一个普通Service组件的其他回调，例如 `onStartCommand()` 会被IntentService自动调用。在IntentService中，要避免重写那些回调。

2)在Manifest文件中定义IntentService

IntentService需要在manifest文件添加相应的条目，将此条目 `<service>` 作为 `<application>` 元素的子元素下进行定义，如下所示：

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name">
    ...
    <!--
        Because android:exported is set to "false",
        the service is only available to this app.
    -->
    <service
        android:name=".RSSPullService"
        android:exported="false"/>
    ...
<application/>
```

`android:name` 属性指明了IntentService的名字。

注意 `<service>` 标签并没有包含任何intent filter。因为发送任务给IntentService的Activity需要使用显式Intent，所以不需要filter。这也意味着只有在同一个app或者其他使用同一个UserID的组件才能够访问到这个Service。

至此，你已经有了一个基本的IntentService类，你可以通过构造Intent对象向它发送操作请求。构造这些对象以及发送它们到你的IntentService的方式，将在接下来的课程中描述。

向后台服务发送任务请求

> 编写:kesenhoo - 原文:http://developer.android.com/training/run-background-service/send-request.html

前一篇文章演示了如何创建一个IntentService类。这次会演示如何通过发送一个Intent来触发IntentService执行任务。这个Intent可以传递一些数据给IntentService。我们可以在Activity或者Fragment的任何时间点发送这个Intent。

创建任务请求并发送到IntentService

为了创建一个任务请求并发送到IntentService。需要先创建一个显式Intent，并将请求数据添加到intent中，然后通过调用 `startService()` 方法把任务请求数据发送到IntentService。

下面的是代码示例：

- 创建一个新的显式Intent用来启动IntentService。

```
/*
 * Creates a new Intent to start the RSSPullService
 * IntentService. Passes a URI in the
 * Intent's "data" field.
 */
mServiceIntent = new Intent(getActivity(), RSSPullService.class);
mServiceIntent.setData(Uri.parse(dataUrl));
```

- 执行 `startService()`

```
// Starts the IntentService
getActivity().startService(mServiceIntent);
```

注意可以在Activity或者Fragment的任何位置发送任务请求。例如，如果你先获取用户输入，您可以从响应按钮单击或类似手势的回调方法里面发送任务请求。

一旦执行了startService()，IntentService在自己本身的 `onHandleIntent()` 方法里面开始执行这个任务，任务结束之后，会自动停止这个Service。

下一步是如何把工作任务的执行结果返回给发送任务的Activity或者Fragment。下节课会演示如何使用BroadcastReceiver来完成这个任务。

报告任务执行状态

> 编写:kesenhoo - 原文:http://developer.android.com/training/run-background-service/report-status.html

这章节会演示如何回传IntentService中执行的任务状态与结果给发送方。 例如，回传任务的执行状态给Activity并进行更新UI。推荐的方式是使用LocalBroadcastManager，这个组件可以限制broadcast intent只在自己的app中进行传递。

利用IntentService 发送任务状态

为了在IntentService中向其他组件发送任务状态，首先创建一个Intent并在data字段中包含需要传递的信息。作为一个可选项，还可以给这个Intent添加一个action与data URI。

下一步，通过执行 `LocalBroadcastManager.sendBroadcast()` 来发送Intent。Intent被发送到任何有注册接受它的组件中。为了获取到LocalBroadcastManager的实例，可以执行getInstance()。代码示例如下：

```
public final class Constants {
    ...
    // Defines a custom Intent action
    public static final String BROADCAST_ACTION =
        "com.example.android.threadsample.BROADCAST";
    ...
    // Defines the key for the status "extra" in an Intent
    public static final String EXTENDED_DATA_STATUS =
        "com.example.android.threadsample.STATUS";
    ...
}
public class RSSPullService extends IntentService {
...
    /*
     * Creates a new Intent containing a Uri object
     * BROADCAST_ACTION is a custom Intent action
     */
    Intent localIntent =
            new Intent(Constants.BROADCAST_ACTION)
            // Puts the status into the Intent
            .putExtra(Constants.EXTENDED_DATA_STATUS, status);
    // Broadcasts the Intent to receivers in this app.
    LocalBroadcastManager.getInstance(this).sendBroadcast(localIntent);
...
}
```

下一步是在发送任务的组件中接收发送出来的broadcast数据。

接收来自IntentService的状态广播

为了接受广播的数据对象，需要使用BroadcastReceiver的子类并实现 `BroadcastReceiver.onReceive()` 的方法，这里可以接收LocalBroadcastManager发出的广播数据。

```
// Broadcast receiver for receiving status updates from the IntentService
private class ResponseReceiver extends BroadcastReceiver
{
    // Prevents instantiation
    private DownloadStateReceiver() {
    }
    // Called when the BroadcastReceiver gets an Intent it's registered to receive
    @
    public void onReceive(Context context, Intent intent) {
...
        /*
         * Handle Intents here.
         */
...
    }
}
```

一旦定义了BroadcastReceiver，也应该定义actions，categories与data用过滤广播。为了实现这些，需要使用IntentFilter。如下所示：

```
// Class that displays photos
public class DisplayActivity extends FragmentActivity {
    ...
    public void onCreate(Bundle stateBundle) {
        ...
        super.onCreate(stateBundle);
        ...
        // The filter's action is BROADCAST_ACTION
        IntentFilter mStatusIntentFilter = new IntentFilter(
                Constants.BROADCAST_ACTION);

        // Adds a data filter for the HTTP scheme
        mStatusIntentFilter.addDataScheme("http");
        ...
```

为了给系统注册这个BroadcastReceiver和IntentFilter，需要通过LocalBroadcastManager执行registerReceiver()的方法。如下所示：

```
// Instantiates a new DownloadStateReceiver
DownloadStateReceiver mDownloadStateReceiver =
        new DownloadStateReceiver();
// Registers the DownloadStateReceiver and its intent filters
LocalBroadcastManager.getInstance(this).registerReceiver(
        mDownloadStateReceiver,
        mStatusIntentFilter);
...
```

一个BroadcastReceiver可以处理多种类型的广播数据。每个广播数据都有自己的ACTION。这个功能使得不用定义多个不同的BroadcastReceiver来分别处理不同的ACTION数据。为BroadcastReceiver定义另外一个IntentFilter，只需要创建一个新的IntentFilter并重复执行registerReceiver()即可。例如:

```
/*
 * Instantiates a new action filter.
 * No data filter is needed.
 */
statusIntentFilter = new IntentFilter(Constants.ACTION_ZOOM_IMAGE);
...
// Registers the receiver with the new filter
LocalBroadcastManager.getInstance(getActivity()).registerReceiver(
        mDownloadStateReceiver,
        mIntentFilter);
```

发送一个广播Intent并不会启动或重启一个Activity。即使是你的app在后台运行，Activity的BroadcastReceiver也可以接收、处理Intent对象。但是这不会迫使你的app进入前台。当你的app不可见时，如果想通知用户一个发生在后台的事件，建议使用Notification。永远不要为了响应一个广播Intent而去启动Activity。

使用CursorLoader在后台加载数据

> 编写:kesenhoo - 原文:http://developer.android.com/training/load-data-background/index.html

从ContentProvider查询你需要显示的数据是比较耗时的。如果你在Activity中直接执行查询的操作,那么有可能导致Activity出现ANR的错误。即使没有发生ANR,用户也容易感知到一个令人烦恼的UI卡顿。为了避免那些问题,你应该在另外一个线程中执行查询的操作,等待查询操作完成,然后再显示查询结果。

通过CursorLoader对象,你可以用一种简单的方式实现异步查询,查询结束时它会和Activity进行重新连接。 CursorLoader不仅仅能够实现在后台查询数据,还能够在查询数据发生变化时自动执行重新查询的操作。

这节课会介绍如何使用CursorLoader来执行一个后台查询数据的操作。在这节课中的演示代码使用的是v4 Support Library中的类。

Demos

ThreadSample

Lessons

- 使用CursorLoader执行查询任务

  学习如何使用CursorLoader在后台执行查询操作。

- 处理CursorLoader查询的结果

  学习如何处理从CursorLoader查询到的数据,以及在loader框架重置CursorLoader时如何解除当前Cursor的引用。

使用CursorLoader执行查询任务

> 编写:kesenhoo - 原文:http://developer.android.com/training/load-data-background/setup-loader.html

CursorLoader通过ContentProvider在后台执行一个异步的查询操作，并且返回数据给调用它的Activity或者FragmentActivity。这使得Activity或者FragmentActivity能够在查询任务正在执行的同时继续与用户进行其他的交互操作。

定义使用CursorLoader的Activity

为了在Activity或者FragmentActivity中使用CursorLoader，它们需要实现 `LoaderCallbacks<Cursor>` 接口。CursorLoader会调用 `LoaderCallbacks<Cursor>` 定义的这些回调方法与Activity进行交互；这节课与下节课会详细介绍每一个回调方法。

例如，下面演示了FragmentActivity如何使用CursorLoader。

```
public class PhotoThumbnailFragment extends FragmentActivity implements
        LoaderManager.LoaderCallbacks<Cursor> {
...
}
```

初始化查询

为了初始化查询，需要调用 `LoaderManager.initLoader()` 。这个方法可以初始化LoaderManager的后台查询框架。你可以在用户输入查询条件之后触发初始化的操作，如果你不需要用户输入数据作为查询条件，你可以在 `onCreate()` 或者 `onCreateView()` 里面触发这个方法。例如：

```
// Identifies a particular Loader being used in this component
private static final int URL_LOADER = 0;
...
/* When the system is ready for the Fragment to appear, this displays
 * the Fragment's View
 */
public View onCreateView(
        LayoutInflater inflater,
        ViewGroup viewGroup,
        Bundle bundle) {
    ...
    /*
     * Initializes the CursorLoader. The URL_LOADER value is eventually passed
     * to onCreateLoader().
     */
    getLoaderManager().initLoader(URL_LOADER, null, this);
    ...
}
```

> **Note:** `getLoaderManager()` 仅仅是在Fragment类中可以直接访问。为了在FragmentActivity中获取到LoaderManager，需要执行 `getSupportLoaderManager()` .

开始查询

一旦后台任务被初始化好，它会执行你实现的回调方法 `onCreateLoader()` 。为了启动查询任务，会在这个方法里面返回CursorLoader。你可以初始化一个空的CursorLoader然后使用它的方法来定义你的查询条件，或者你可以在初始化CursorLoader对象的时候就同时定义好查询条件：

```
/*
* Callback that's invoked when the system has initialized the Loader and
* is ready to start the query. This usually happens when initLoader() is
* called. The loaderID argument contains the ID value passed to the
* initLoader() call.
*/
@Override
public Loader<Cursor> onCreateLoader(int loaderID, Bundle bundle)
{
    /*
     * Takes action based on the ID of the Loader that's being created
     */
    switch (loaderID) {
        case URL_LOADER:
            // Returns a new CursorLoader
            return new CursorLoader(
                        getActivity(),   // Parent activity context
                        mDataUrl,        // Table to query
                        mProjection,     // Projection to return
                        null,            // No selection clause
                        null,            // No selection arguments
                        null             // Default sort order
        );
        default:
            // An invalid id was passed in
            return null;
    }
}
```

一旦后台查询任务获取到了这个Loader对象，就开始在后台执行查询的任务。当查询完成之后，会执行 `onLoadFinished()` 这个回调函数，关于这些内容会在下一节讲到。

处理查询的结果

> 编写:kesenhoo - 原文:http://developer.android.com/training/load-data-background/handle-results.html

正如前面一节课讲到的，你应该在 onCreateLoader()的回调里面使用CursorLoader执行加载数据的操作。Loader查询完后会调用Activity或者FragmentActivity的LoaderCallbacks.onLoadFinished()将结果回调回来。这个回调方法的参数之一是Cursor，它包含了查询的数据。你可以使用Cursor对象来更新需要显示的数据或者进行下一步的处理。

除了onCreateLoader()与onLoadFinished()，你也需要实现onLoaderReset()。这个方法在CursorLoader检测到Cursor上的数据发生变化的时候会被触发。当数据发生变化时，系统也会触发重新查询的操作。

处理查询结果

为了显示CursorLoader返回的Cursor数据，需要使用实现AdapterView的视图组件，，并为这个组件绑定一个实现了CursorAdapter的Adapter。系统会自动把Cursor中的数据显示到View上。

你可以在显示数据之前建立View与Adapter的关联。然后在onLoadFinished()的时候把Cursor与Adapter进行绑定。一旦你把Cursor与Adapter进行绑定之后，系统会自动更新View。当Cursor上的内容发生改变的时候，也会触发这些操作。

例如:

```
public String[] mFromColumns = {
    DataProviderContract.IMAGE_PICTURENAME_COLUMN
};
public int[] mToFields = {
    R.id.PictureName
};
// Gets a handle to a List View
ListView mListView = (ListView) findViewById(R.id.dataList);
/*
 * Defines a SimpleCursorAdapter for the ListView
 *
 */
SimpleCursorAdapter mAdapter =
    new SimpleCursorAdapter(
            this,                // Current context
            R.layout.list_item,  // Layout for a single row
            null,                // No Cursor yet
            mFromColumns,        // Cursor columns to use
            mToFields,           // Layout fields to use
            0                    // No flags
    );
// Sets the adapter for the view
mListView.setAdapter(mAdapter);
...
/*
 * Defines the callback that CursorLoader calls
 * when it's finished its query
 */
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    ...
    /*
     * Moves the query results into the adapter, causing the
     * ListView fronting this adapter to re-display
     */
    mAdapter.changeCursor(cursor);
}
```

删除废旧的Cursor引用

当Cursor失效的时候，CursorLoader会被重置。这通常发生在Cursor相关的数据改变的时候。在重新执行查询操作之前，系统会执行你的onLoaderReset()回调方法。在这个回调方法中，你应该删除当前Cursor上的所有数据，避免发生内存泄露。一旦onLoaderReset()执行结束，CursorLoader就会重新执行查询操作。

例如:

```
/*
 * Invoked when the CursorLoader is being reset. For example, this is
 * called if the data in the provider changes and the Cursor becomes stale.
 */
@Override
public void onLoaderReset(Loader<Cursor> loader) {

    /*
     * Clears out the adapter's reference to the Cursor.
     * This prevents memory leaks.
     */
    mAdapter.changeCursor(null);
}
```

管理设备的唤醒状态

> 编写:jdneo，lttowq - 原文:http://developer.android.com/training/scheduling/index.html

当一个Android设备闲置时，首先它的屏幕将会变暗，然后关闭屏幕，最后关闭CPU。 这样可以防止设备的电量被迅速消耗殆尽。但是，有时候也会存在一些特例：

- 例如游戏或视频应用需要保持屏幕常亮；
- 其它应用也许不需要屏幕常亮，但或许会需要CPU保持运行，直到某个关键操作结束。

这节课描述如何在必要的时候保持设备唤醒，同时又不会过多消耗它的电量。

Demos

Scheduler.zip

Lessons

保持设备唤醒

学习如何在必要的时候保持屏幕和CPU唤醒，同时减少对电池寿命的影响。

调度重复闹钟

对于那些发生在应用生命周期之外的操作，学习如何使用重复闹钟对它们进行调度，即使该应用没有运行或者设备处于睡眠状态。

保持设备唤醒

> 编写:jdneo - 原文:http://developer.android.com/training/scheduling/wakelock.html

为了避免电量过度消耗，Android设备会在被闲置之后迅速进入睡眠状态。然而有时候应用会需要唤醒屏幕或者是唤醒CPU并且保持它们的唤醒状态，直至一些任务被完成。

想要做到这一点，所采取的方法依赖于应用的具体需求。但是通常来说，我们应该使用最轻量级的方法，减小其对系统资源的影响。在接下来的部分中，我们将会描述在设备默认的睡眠行为与应用的需求不相符合的情况下，我们应该如何进行对应的处理。

保持屏幕常亮

某些应用需要保持屏幕常亮，比如游戏与视频应用。最好的方式是在你的Activity中（且仅在Activity中，而不是在Service或其他应用组件里）使用FLAG_KEEP_SCREEN_ON属性，例如：

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
  }
```

该方法的优点与唤醒锁（Wake Locks）不同（唤醒锁的内容在本章节后半部分），它不需要任何特殊的权限，系统会正确地 管理应用之间的切换，且不必关心释放资源的问题。

另外一种方法是在应用的XML布局文件里，使用android:keepScreenOn属性:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:keepScreenOn="true">
    ...
</RelativeLayout>
```

使用 `android:keepScreenOn="true"` 与使用FLAG_KEEP_SCRRE_ON等效。你可以选择最适合你的应用的方法。在Activity中通过代码设置常亮标识的优点在于：你可以通过代码动态清除这个标示，从而使屏幕可以关闭。

> **Notes**：除非你不再希望正在运行的应用长时间点亮屏幕（例如：在一定时间无操作发生后，你想要将屏幕关闭），否则你是不需要清除FLAG_KEEP_SCRRE_ON 标识的。WindowManager会在应用进入后台或者返回前台时，正确管理屏幕的点亮或者关闭。但是如果你想要显式地清除这一标识，从而使得屏幕能够关闭，可以使用 `getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)` 方法。

保持CPU运行

如果你需要在设备睡眠之前，保持CPU运行来完成一些工作，你可以使用PowerManager系统服务中的唤醒锁功能。唤醒锁允许应用控制设备的电源状态。

创建和保持唤醒锁会对设备的电源寿命产生巨大影响。因此你应该仅在你确实需要时使用唤醒锁，且使用的时间应该越短越好。如果想要在Activity中使用唤醒锁就显得没有必要了。如上所述，可以在Activity中使用FLAG_KEEP_SCRRE_ON让屏幕保持常亮。

使用唤醒锁的一种合理情况可能是：一个后台服务需要在屏幕关闭时利用唤醒锁保持CPU运行。再次强调，应该尽可能规避使用该方法，因为它会影响到电池寿命。

> 不必使用唤醒锁的情况：
>
> 1. 如果你的应用正在执行一个HTTP长连接的下载任务，可以考虑使用DownloadManager。
> 2. 如果你的应用正在从一个外部服务器同步数据，可以考虑创建一个SyncAdapter
> 3. 如果你的应用需要依赖于某些后台服务，可以考虑使用RepeatingAlarm或者Google Cloud Messaging，以此每隔特定的时间，将这些服务激活。

为了使用唤醒锁，首先需要在应用的Manifest清单文件中增加WAKE_LOCK权限：

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

如果你的应用包含一个BroadcastReceiver并使用Service来完成一些工作，你可以通过WakefulBroadcastReceiver管理你唤醒锁。后续章节中将会提到，这是一种推荐的方法。如果你的应用不满足上述情况，可以使用下面的方法直接设置唤醒锁：

```
PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
Wakelock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
        "MyWakelockTag");
wakeLock.acquire();
```

可以调用 `wakelock.release()` 来释放唤醒锁。当应用使用完毕时，应该释放该唤醒锁，以避免电量过度消耗。

使用WakefulBroadcastReceiver

你可以将BroadcastReceiver和Service结合使用，以此来管理后台任务的生命周期。WakefulBroadcastReceiver是一种特殊的BroadcastReceiver，它专注于创建和管理应用的PARTIAL_WAKE_LOCK。WakefulBroadcastReceiver会将任务交付给Service（一般会是一个IntentService），同时确保设备在此过程中不会进入睡眠状态。如果在该过程当中没有保持住唤醒锁，那么还没等任务完成，设备就有可能进入睡眠状态了。其结果就是：应用可能会在未来的某一个时间节点才把任务完成，这显然不是你所期望的。

要使用WakefulBroadcastReceiver，首先在Manifest文件添加一个标签：

```
<receiver android:name=".MyWakefulReceiver"></receiver>
```

下面的代码通过 `startWakefulService()` 启动 `MyIntentService` 。该方法和 `startService()` 类似，除了WakeflBroadcastReceiver会在Service启动后将唤醒锁保持住。传递给 `startWakefulService()` 的Intent会携带有一个Extra数据，用来标识唤醒锁。

```
public class MyWakefulReceiver extends WakefulBroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        // Start the service, keeping the device awake while the service is
        // launching. This is the Intent to deliver to the service.
        Intent service = new Intent(context, MyIntentService.class);
        startWakefulService(context, service);
    }
}
```

当Service结束之后，它会调用 `MyWakefulReceiver.completeWakefulIntent()` 来释放唤醒锁。 `completeWakefulIntent()` 方法中的Intent参数是和WakefulBroadcastReceiver传递进来的Intent参数一致的：

```
public class MyIntentService extends IntentService {
    public static final int NOTIFICATION_ID = 1;
    private NotificationManager mNotificationManager;
    NotificationCompat.Builder builder;
    public MyIntentService() {
        super("MyIntentService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        Bundle extras = intent.getExtras();
        // Do the work that requires your app to keep the CPU running.
        // ...
        // Release the wake lock provided by the WakefulBroadcastReceiver.
        MyWakefulReceiver.completeWakefulIntent(intent);
    }
}
```

调度重复的闹钟

> 编写:jdneo - 原文:http://developer.android.com/training/scheduling/alarms.html

闹钟（基于AlarmManager类）给予你一种在应用使用期之外执行与时间相关的操作的方法。你可以使用闹钟初始化一个长时间的操作，例如每天开启一次后台服务，下载当日的天气预报。

闹钟具有如下特性：

- 允许你通过预设时间或者设定某个时间间隔，来触发Intent；
- 你可以将它与BroadcastReceiver相结合，来启动服务并执行其他操作；
- 可在应用范围之外执行，所以你可以在你的应用没有运行或设备处于睡眠状态的情况下，使用它来触发事件或行为；
- 帮助你的应用最小化资源需求，你可以使用闹钟调度你的任务，来替代计时器或者长时间连续运行的后台服务。

> **Note**：对于那些需要确保在应用使用期之内发生的定时操作，可以使用闹钟替代使用Handler结合Timer与Thread的方法。因为它可以让Android系统更好地统筹系统资源。

权衡利弊

重复闹钟的机制比较简单，没有太多的灵活性。它对于你的应用来说或许不是一种最好的选择，特别是当你想要触发网络操作的时候。设计不佳的闹钟会导致电量快速耗尽，而且会对服务端产生巨大的负荷。

当我们从服务端同步数据时，往往会在应用不被使用的时候时被唤醒触发执行某些操作。此时你可能希望使用重复闹钟。但是如果存储数据的服务端是由你控制的，使用Google Cloud Messaging（GCM）结合sync adapter是一种更好解决方案。SyncAdapter提供的任务调度选项和AlarmManager基本相同，但是它能提供更多的灵活性。比如：同步的触发可能基于一条“新数据”提示消息，而消息的产生可以基于服务器或设备，用户的操作（或者没有操作），每天的某一时刻等等。

最佳实践方法

在设计重复闹钟过程中，你所做出的每一个决定都有可能影响到你的应用将会如何使用系统资源。例如，我们假想一个会从服务器同步数据的应用。同步操作基于的是时钟时间，具体来说，每一个应用的实例会在下午十一点整进行同步，巨大的服务器负荷会导致服务器响应时间变长，甚至拒绝服务。因此在我们使用闹钟时，请牢记下面的最佳实践建议：

- 对任何由重复闹钟触发的网络请求添加一定的随机性（抖动）：
    - 在闹钟触发时做一些本地任务。“本地任务”指的是任何不需要访问服务器或者从服务器获取数据的任务；
    - 同时对于那些包含有网络请求的闹钟，在调度时机上增加一些随机性。
- 尽量让你的闹钟频率最小；
- 如果不是必要的情况，不要唤醒设备（这一点与闹钟的类型有关，本节课后续部分会提到）；
- 触发闹钟的时间不必过度精确；尽量使用 `setInexactRepeating()` 方法替代 `setRepeating()` 方法。当你使用 `setInexactRepeating()` 方法时，Android系统会集中多个应用的重复闹钟同步请求，并一起触发它们。这可以减少系统将设备唤醒的总次数，以此减少电量消耗。从Android 4.4（API Level19）开始，所有的重复闹钟都将是非精确型的。注意虽然 `setInexactRepeating()` 是 `setRepeating()` 的改进版本，它依然可能会导致每一个应用的实例在某一时间段内同时访问服务器，造成服务器负荷过重。因此如之前所述，对于网络请求，我们需要为闹钟的触发时机增加随机性。
- 尽量避免让闹钟基于时钟时间。

想要在某一个精确时刻触发重复闹钟是比较困难的。我们应该尽可能使用ELAPSED_REALTIME。不同的闹钟类型会在本节课后半部分展开。

设置重复闹钟

如上所述，对于定期执行的任务或者数据查询而言，使用重复闹钟是一个不错的选择。它具有下列属性：

- 闹钟类型（后续章节中会展开讨论）；
- 触发时间。如果触发时间是过去的某个时间点，闹钟会立即被触发；
- 闹钟间隔时间。例如，一天一次，每小时一次，每五秒一次，等等；
- 在闹钟被触发时才被发出的Pending Intent。如果你为同一个Pending Intent设置了另一个闹钟，那么它会将第一个闹钟覆盖。

选择闹钟类型

使用重复闹钟要考虑的第一件事情是闹钟的类型。

闹钟类型有两大类： `ELAPSED_REALTIME` 和 `REAL_TIME_CLOCK` （RTC）。 `ELAPSED_REALTIME` 从系统启动之后开始计算， `REAL_TIME_CLOCK` 使用的是世界统一时间（UTC）。也就是说由于 `ELAPSED_REALTIME` 不受地区和时区的影响，所以它适合于基于时间差的闹钟（例如一个每过30秒触发一次的闹钟）。 `REAL_TIME_CLOCK` 适合于那些依赖于地区位置的闹钟。

两种类型的闹钟都还有一个唤醒（ `WAKEUP` ）版本，也就是可以在设备屏幕关闭的时候唤醒CPU。这可以确保闹钟会在既定的时间被激活，这对于那些实时性要求比较高的应用（比如含有一些对执行时间有要求的操作）来说非常有效。如果你没有使用唤醒版本的闹钟，那么所有的重复闹钟会在下一次设备被唤醒时被激活。

如果你只是简单的希望闹钟在一个特定的时间间隔被激活（例如每半小时一次），那么你可以使用任意一种 `ELAPSED_REALTIME` 类型的闹钟，通常这会是一个更好的选择。

如果你的闹钟是在每一天的特定时间被激活，那么你可以选择 `REAL_TIME_CLOCK` 类型的闹钟。不过需要注意的是，这个方法会有一些缺陷——如果地区发生了变化，应用可能无法做出正确的改变；另外，如果用户改变了设备的时间设置，这可能会造成应用产生预期之外的行为。使用 `REAL_TIME_CLOCK` 类型的闹钟还会有精度的问题，因此我们建议你尽可能使用 `ELAPSED_REALTIME` 类型。

下面列出闹钟的具体类型：

- ELAPSED_REALTIME：从设备启动之后开始算起，度过了某一段特定时间后，激活Pending Intent，但不会唤醒设备。其中设备睡眠的时间也会包含在内。
- ELAPSED_REALTIME_WAKEUP：从设备启动之后开始算起，度过了某一段特定时间后唤醒设备。
- RTC：在某一个特定时刻激活Pending Intent，但不会唤醒设备。
- RTC_WAKEUP：在某一个特定时刻唤醒设备并激活Pending Intent。

ELAPSED_REALTIME_WAKEUP案例

下面是使用ELAPSED_REALTIME_WAKEUP的例子。

每隔在30分钟后唤醒设备以激活闹钟：

```
// Hopefully your alarm will have a lower frequency than this!
alarmMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        AlarmManager.INTERVAL_HALF_HOUR,
        AlarmManager.INTERVAL_HALF_HOUR, alarmIntent);
```

在一分钟后唤醒设备并激活一个一次性（无重复）闹钟：

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() +
        60 * 1000, alarmIntent);
```

RTC案例

下面是使用RTC_WAKEUP的例子。

在大约下午2点唤醒设备并激活闹钟，并不断重复：

```
// Set the alarm to start at approximately 2:00 p.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 14);

// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_DAY, alarmIntent);
```

让设备精确地在上午8点半被唤醒并激活闹钟，自此之后每20分钟唤醒一次：

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

// Set the alarm to start at 8:30 a.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 8);
calendar.set(Calendar.MINUTE, 30);

// setRepeating() lets you specify a precise custom interval--in this case,
// 20 minutes.
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
        1000 * 60 * 20, alarmIntent);
```

决定闹钟的精确度

如上所述，创建闹钟的第一步是要选择闹钟的类型，然后你需要决定闹钟的精确度。对于大多数应用而言，`setInexactRepeating()` 会是一个正确的选择。当你使用该方法时，Android系统会集中多个应用的重复闹钟同步请求，并一起触发它们。这样可以减少电量的损耗。

对于另一些实时性要求较高的应用——例如，闹钟需要精确地在上午8点半被激活，并且自此之后每隔1小时激活一次——那么可以使用 `setRepeating()` 。不过你应该尽量避免使用精确的闹钟。

使用 `setRepeating()` 时，你可以制定一个自定义的时间间隔，但在使用 `setInexactRepeating()` 时不支持这么做。此时你只能选择一些时间间隔常量，例如：INTERVAL_FIFTEEN_MINUTES ，INTERVAL_DAY等。完整的常量列表，可以查看AlarmManager。

取消闹钟

你可能希望在应用中添加取消闹钟的功能。要取消闹钟，可以调用AlarmManager的 `cancel()` 方法，并把你不想激活的PendingIntent传递进去，例如：

```
// If the alarm has been set, cancel it.
if (alarmMgr!= null) {
    alarmMgr.cancel(alarmIntent);
}
```

在设备启动后启用闹钟

默认情况下，所有的闹钟会在设备关闭时被取消。要防止闹钟被取消，你可以让你的应用在用户重启设备后自动重启一个重复闹钟。这样可以让AlarmManager继续执行它的工作，且不需要用户手动重启闹钟。

具体步骤如下：

1.在应用的Manifest文件中设置RECEIVE_BOOT_CMPLETED权限，这将允许你的应用接收系统启动完成后发出的ACTION_BOOT_COMPLETED广播（只有在用户至少将你的应用启动了一次后，这样做才有效）：

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

2.实现BoradcastReceiver用于接收广播：

```
public class SampleBootReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("android.intent.action.BOOT_COMPLETED")) {
            // Set the alarm here.
        }
    }
}
```

3.在你的Manifest文件中添加一个接收器，其Intent-Filter接收ACTION_BOOT_COMPLETED这一Action：

```
<receiver android:name=".SampleBootReceiver"
        android:enabled="false">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"></action>
    </intent-filter>
</receiver>
```

注意Manifest文件中，对接收器设置了 `android:enabled="false"` 属性。这意味着除非应用显式地启用它，不然该接收器将不被调用。这可以防止接收器被不必要地调用。你可以像下面这样启动接收器（比如用户设置了一个闹钟）：

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class);
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP);
```

一旦你像上面那样启动了接收器，它将一直保持启动状态，即使用户重启了设备也不例外。换句话说，通过代码设置的启用配置将会覆盖掉Manifest文件中的现有配置，即使重启也不例外。接收器将保持启动状态，直到你的应用将其禁用。你可以像下面这样禁用接收器（比如用户取消了一个闹钟）：

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class);
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP);
```

性能优化

> 编写:kesenhoo - 原文:http://developer.android.com/training/best-performance.html

下面的这些课程会介绍如何提升应用的性能，如何尽量减少电量的消耗。

管理应用的内存

如何减少内存的占用。

代码性能优化建议

如何提高应用的响应性与电池的使用效率。

提升Layout的性能

如何提升UI的性能。

优化电池寿命

如何优化电量的消耗。

多线程操作

如何通过多线程分折任务来提高程序性能。

避免出现程序无响应ANR

如何避免ANR。

JNI技巧

如何高效的使用JNI。

SMP Primer for Android

优化多核处理器架构下的Android程序。

管理应用的内存

> 编写:kesenhoo - 原文:http://developer.android.com/training/articles/memory.html

Random Access Memory(RAM)在任何软件开发环境中都是一个很宝贵的资源。这一点在物理内存通常很有限的移动操作系统上，显得尤为突出。尽管Android的Dalvik虚拟机扮演了常规的垃圾回收的角色，但这并不意味着你可以忽视app的内存分配与释放的时机与地点。

为了GC能够从app中及时回收内存，我们需要注意避免内存泄露(通常由于在全局成员变量中持有对象引用而导致)并且在适当的时机(下面会讲到的lifecycle callbacks)来释放引用对象。对于大多数app来说，Dalvik的GC会自动把离开活动线程的对象进行回收。

这篇文章会解释Android是如何管理app的进程与内存分配，以及在开发Android应用的时候如何主动的减少内存的使用。关于Java的资源管理机制，请参考其它书籍或者线上材料。如果你正在寻找如何分析你的内存使用情况的文章，请参考这里Investigating Your RAM Usage。

第1部分: Android是如何管理内存的

Android并没有为内存提供交换区(Swap space)，但是它有使用paging与memory-mapping(mmapping)的机制来管理内存。这意味着任何你修改的内存(无论是通过分配新的对象还是去访问mmaped pages中的内容)都会贮存在RAM中，而且不能被paged out。因此唯一完整释放内存的方法是释放那些你可能hold住的对象的引用，当这个对象没有被任何其他对象所引用的时候，它就能够被GC回收了。只有一种例外是：如果系统想要在其他地方重用这个对象。

1) 共享内存

Android通过下面几个方式在不同的进程中来实现共享RAM:

- 每一个app的进程都是从一个被叫做**Zygote**的进程中fork出来的。Zygote进程在系统启动并且载入通用的framework的代码与资源之后开始启动。为了启动一个新的程序进程，系统会fork Zygote进程生成一个新的进程，然后在新的进程中加载并运行app的代码。这使得大多数的RAM pages被用来分配给framework的代码，同时使得RAM资源能够在应用的所有进程中进行共享。

- 大多数static的数据被mmapped到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享，而且使得它能够在需要的时候被paged out。例如下面几种static的数据:
    - Dalvik 代码 (放在一个预链接好的 .odex 文件中以便直接mapping)
    - App resources (通过把资源表结构设计成便于mmapping的数据结构，另外还可以通

    过把APK中的文件做aligning的操作来优化)
    - 传统项目元素，比如 .so 文件中的本地代码.
- 在很多情况下，Android通过显式的分配共享内存区域(例如ashmem或者gralloc)来实现一些动态RAM区域能够在不同进程间进行共享。例如，window surfaces在app与screen compositor之间使用共享的内存，cursor buffers在content provider与client之间使用共享的内存。

关于如何查看app所使用的共享内存，请查看Investigating Your RAM Usage

2) 分配与回收内存

这里有下面几点关于Android如何分配与回收内存的事实：

- 每一个进程的Dalvik heap都有一个受限的虚拟内存范围。这就是逻辑上讲的heap size，它可以随着需要进行增长，但是会有一个系统为它所定义的上限。
- 逻辑上讲的heap size和实际物理上使用的内存数量是不等的，Android会计算一个叫做Proportional Set Size(PSS)的值，它记录了那些和其他进程进行共享的内存大小。（假设共享内存大小是10M，一共有20个Process在共享使用，根据权重，可能认为其中有0.3M才能真正算是你的进程所使用的）
- Dalvik heap与逻辑上的heap size不吻合，这意味着Android并不会去做heap中的碎片整理用来关闭空闲区域。Android仅仅会在heap的尾端出现不使用的空间时才会做收缩逻辑heap size大小的动作。但是这并不是意味着被heap所使用的物理内存大小不能被收缩。在垃圾回收之后，Dalvik会遍历heap并找出不使用的pages，然后使用madvise(系统调用)把那些pages返回给kernal。因此，成对的allocations与deallocations大块的数据可以使得物理内存能够被正常的回收。然而，回收碎片化的内存则会使得效率低下很多，因为那些碎片化的分配页面也许会被其他地方所共享到。

3) 限制应用的内存

为了维持多任务的功能环境，Android为每一个app都设置了一个硬性的heap size限制。准确的heap size限制会因为不同设备的不同RAM大小而各有差异。如果你的app已经到了heap的限制大小并且再尝试分配内存的话，会引起 `OutOfMemoryError` 的错误。

在一些情况下，你也许想要查询当前设备的heap size限制大小是多少，然后决定cache的大小。可以通过 `getMemoryClass()` 来查询。这个方法会返回一个整数，表明你的应用的heap size限制是多少Mb(megabates)。

4) 切换应用

Android并不会在用户切换不同应用时候做交换内存的操作。Android会把那些不包含foreground组件的进程放到LRU cache中。例如，当用户刚开始启动了一个应用，系统会为它创建了一个进程，但是当用户离开这个应用，此进程并不会立即被销毁。系统会把这个进程

放到cache中，如果用户后来再回到这个应用，此进程就能够被完整恢复，从而实现应用的快速切换。

如果你的应用中有一个被缓存的进程，这个进程会占用暂时不需要使用到的内存，这个暂时不需要使用的进程，它被保留在内存中，这会对系统的整体性能有影响。因此当系统开始进入低内存状态时，它会由系统根据LRU的规则与其他因素选择综合考虑之后决定杀掉某些进程，为了保持你的进程能够尽可能长久的被缓存，请参考下面的章节学习何时释放你的引用。

对于那些不在foreground的进程，Android是如何决定kill掉哪一类进程的问题，请参考Processes and Threads.

第2部分: 你的应用该如何管理内存

你应该在开发过程的每一个阶段都考虑到RAM的有限性，甚至包括在开始编写代码之前的设计阶段就应该考虑到RAM的限制性。我们可以使用多种设计与实现方式，他们有着不同的效率，即使这些方式只是相同技术的不断组合与演变。

为了使得你的应用性能效率更高，你应该在设计与实现代码时，遵循下面的技术要点。

1) 珍惜Services资源

如果你的应用需要在后台使用service，除非它被触发并执行一个任务，否则其他时候service都应该是停止状态。另外需要注意当这个service完成任务之后因为停止service失败而引起的内存泄漏。

当你启动一个service，系统会倾向为了保留这个service而一直保留service所在的进程。这使得进程的运行代价很高，因为系统没有办法把service所占用的RAM空间腾出来让给其他组件，另外service还不能被paged out。这减少了系统能够存放到LRU缓存当中的进程数量，它会影响app之间的切换效率。它甚至会导致系统内存使用不稳定，从而无法继续保持住所有目前正在运行的service。

限制你的service的最好办法是使用IntentService， 它会在处理完交代给它的intent任务之后尽快结束自己。更多信息，请阅读Running in a Background Service.

当一个Service已经不再需要的时候还继续保留它，这对Android应用的内存管理来说是最糟糕的错误之一。因此千万不要贪婪的使得一个Service持续保留。不仅仅是因为它会使得你的应用因为RAM空间的不足而性能糟糕，还会使得用户发现那些有着常驻后台行为的应用并且可能卸载它。

2) 当UI隐藏时释放内存

当用户切换到其它应用并且你的应用 UI不再可见时，你应该释放你的应用UI上所占用的所有内存资源。在这个时候释放UI资源可以显著的增加系统缓存进程的能力，它会对用户体验有着很直接的影响。

为了能够接收到用户离开你的UI时的通知，你需要实现Activtiy类里面的 `onTrimMemory()` 回调方法。你应该使用这个方法来监听到 `TRIM_MEMORY_UI_HIDDEN` 级别的回调，此时意味着你的UI已经隐藏，你应该释放那些仅仅被你的UI使用的资源。

请注意：你的应用仅仅会在所有UI组件的被隐藏的时候接收到 `onTrimMemory()` 的回调并带有参数 `TRIM_MEMORY_UI_HIDDEN` 。这与onStop()的回调是不同的，onStop会在activity的实例隐藏时会执行，例如当用户从你的app的某个activity跳转到另外一个activity时前面activity的onStop()会被执行。因此你应该实现onStop回调，并且在此回调里面释放activity的资源，例如释放网络连接，注销监听广播接收者。除非接收到onTrimMemory(TRIM_MEMORY_UI_HIDDEN))的回调，否者你不应该释放你的UI资源。这确保了用户从其他activity切回来时，你的UI资源仍然可用，并且可以迅速恢复activity。

3) 当内存紧张时释放部分内存

在你的app生命周期的任何阶段，onTrimMemory的回调方法同样可以告诉你整个设备的内存资源已经开始紧张。你应该根据onTrimMemory回调中的内存级别来进一步决定释放哪些资源。

- TRIM_MEMORY_RUNNING_MODERATE：你的app正在运行并且不会被列为可杀死的。但是设备此时正运行于低内存状态下，系统开始触发杀死LRU Cache中的Process的机制。
- TRIM_MEMORY_RUNNING_LOW：你的app正在运行且没有被列为可杀死的。但是设备正运行于更低内存的状态下，你应该释放不用的资源用来提升系统性能（但是这也会直接影响到你的app的性能）。
- TRIM_MEMORY_RUNNING_CRITICAL：你的app仍在运行，但是系统已经把LRU Cache中的大多数进程都已经杀死，因此你应该立即释放所有非必须的资源。如果系统不能回收到足够的RAM数量，系统将会清除所有的LRU缓存中的进程，并且开始杀死那些之前被认为不应该杀死的进程，例如那个包含了一个运行态Service的进程。

同样，当你的app进程正在被cached时，你可能会接受到从onTrimMemory()中返回的下面的值之一:

- TRIM_MEMORY_BACKGROUND: 系统正运行于低内存状态并且你的进程正处于LRU缓存名单中最不容易杀掉的位置。尽管你的app进程并不是处于被杀掉的高危险状态，系统可能已经开始杀掉LRU缓存中的其他进程了。你应该释放那些容易恢复的资源，以便于你的进程可以保留下来，这样当用户回退到你的app的时候才能够迅速恢复。
- TRIM_MEMORY_MODERATE: 系统正运行于低内存状态并且你的进程已经已经接近LRU名单的中部位置。如果系统开始变得更加内存紧张，你的进程是有可能被杀死的。
- TRIM_MEMORY_COMPLETE: 系统正运行与低内存的状态并且你的进程正处于LRU名

单中最容易被杀掉的位置。你应该释放任何不影响你的app恢复状态的资源。

因为onTrimMemory()的回调是在**API 14**才被加进来的，对于老的版本，你可以使用onLowMemory)回调来进行兼容。onLowMemory相当与 `TRIM_MEMORY_COMPLETE` 。

> **Note:** 当系统开始清除LRU缓存中的进程时，尽管它首先按照LRU的顺序来操作，但是它同样会考虑进程的内存使用量。因此消耗越少的进程则越容易被留下来。

4) 检查你应该使用多少的内存

正如前面提到的，每一个Android设备都会有不同的RAM总大小与可用空间，因此不同设备为app提供了不同大小的heap限制。你可以通过调用getMemoryClass())来获取你的app的可用heap大小。如果你的app尝试申请更多的内存，会出现 `OutOfMemory` 的错误。

在一些特殊的情景下，你可以通过在manifest的application标签下添加 `largeHeap=true` 的属性来声明一个更大的heap空间。如果你这样做，你可以通过getLargeMemoryClass())来获取到一个更大的heap size。

然而，能够获取更大heap的设计本意是为了一小部分会消耗大量RAM的应用(例如一个大图片的编辑应用)。不要轻易的因为你需要使用大量的内存而去请求一个大的**heap size**。只有当你清楚的知道哪里会使用大量的内存并且为什么这些内存必须被保留时才去使用large heap. 因此请尽量少使用large heap。使用额外的内存会影响系统整体的用户体验，并且会使得GC的每次运行时间更长。在任务切换时，系统的性能会变得大打折扣。

另外, large heap并不一定能够获取到更大的heap。在某些有严格限制的机器上，large heap的大小和通常的heap size是一样的。因此即使你申请了large heap，你还是应该通过执行getMemoryClass()来检查实际获取到的heap大小。

5) 避免**bitmaps**的浪费

当你加载一个bitmap时，仅仅需要保留适配当前屏幕设备分辨率的数据即可，如果原图高于你的设备分辨率，需要做缩小的动作。请记住，增加bitmap的尺寸会对内存呈现出2次方的增加，因为X与Y都在增加。

> **Note:**在Android 2.3.x (API level 10)及其以下, bitmap对象的pixel data是存放在native内存中的，它不便于调试。然而，从Android 3.0(API level 11)开始，bitmap pixel data是分配在你的app的Dalvik heap中, 这提升了GC的工作效率并且更加容易Debug。因此如果你的app使用bitmap并在旧的机器上引发了一些内存问题，切换到3.0以上的机器上进行Debug。

6) 使用优化的数据容器

利用Android Framework里面优化过的容器类，例如SparseArray, SparseBooleanArray, 与LongSparseArray。 通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效在于他们避免了对key与value的autobox自动装箱，并且避免了装箱后的解箱。

7) 请注意内存开销

对你所使用的语言与库的成本与开销有所了解，从开始到结束，在设计你的app时谨记这些信息。通常，表面上看起来无关痛痒(innocuous)的事情也许实际上会导致大量的开销。例如：

- Enums的内存消耗通常是static constants的2倍。你应该尽量避免在Android上使用enums。
- 在Java中的每一个类(包括匿名内部类)都会使用大概500 bytes。
- 每一个类的实例花销是12-16 bytes。
- 往HashMap添加一个entry需要额一个额外占用的32 bytes的entry对象。

8) 请注意代码“抽象”

通常，开发者使用抽象作为"好的编程实践"，因为抽象能够提升代码的灵活性与可维护性。然而，抽象会导致一个显著的开销：通常他们需要同等量的代码用于可执行。那些代码会被map到内存中。因此如果你的抽象没有显著的提升效率，应该尽量避免他们。

9) 为序列化的数据使用nano protobufs

Protocol buffers是由Google为序列化结构数据而设计的，一种语言无关，平台无关，具有良好扩展性的协议。类似XML，却比XML更加轻量，快速，简单。如果你需要为你的数据实现协议化，你应该在客户端的代码中总是使用nano protobufs。通常的协议化操作会生成大量繁琐的代码，这容易给你的app带来许多问题：增加RAM的使用量，显著增加APK的大小，更慢的执行速度，更容易达到DEX的字符限制。

关于更多细节，请参考protobuf readme的"Nano version"章节。

10) 避免使用依赖注入框架

使用类似Guice或者RoboGuice等framework injection包是很有效的，因为他们能够简化你的代码。

> Notes：RoboGuice 2 通过依赖注入改变代码风格，让Android开发时的体验更好。你在调用 `getIntent().getExtras()` 时经常忘记检查 null 吗？RoboGuice 2 可以帮你做。你认为将 `findViewById()` 的返回值强制转换成 TextView 是本不必要的工作吗？RoboGuice 2 可以帮你。RoboGuice 2 把这些需要猜测性的工作移到Android开发以外去了。RoboGuice 2 会负责注入你的 View, Resource, System Service或者其他对象等等类似的细节。

然而，那些框架会通过扫描你的代码执行许多初始化的操作，这会导致你的代码需要大量的RAM来mapping代码，而且mapped pages会长时间的被保留在RAM中。

11) 谨慎使用第三方libraries

很多开源的library代码都不是为移动网络环境而编写的，如果运用在移动设备上，，这样的效率并不高。当你决定使用一个第三方library的时候，你应该针对移动网络做繁琐的迁移与维护的工作。

即使是针对Android而设计的library，也可能是很危险的，因为每一个library所做的事情都是不一样的。例如，其中一个lib使用的是nano protobufs, 而另外一个使用的是micro protobufs。那么这样，在你的app里面就有2种protobuf的实现方式。这样的冲突同样可能发生在输出日志，加载图片，缓存等等模块里面。

同样不要陷入为了1个或者2个功能而导入整个library的陷阱。如果没有一个合适的库与你的需求相吻合，你应该考虑自己去实现，而不是导入一个大而全的解决方案。

12) 优化整体性能

官方有列出许多优化整个app性能的文章：Best Practices for Performance。这篇文章就是其中之一。有些文章是讲解如何优化app的CPU使用效率，有些是如何优化app的内存使用效率。

你还应该阅读optimizing your UI来为layout进行优化。同样还应该关注lint工具所提出的建议，进行优化。

13) 使用ProGuard来剔除不需要的代码

ProGuard能够通过移除不需要的代码，重命名类，域与方法等方对代码进行压缩，优化与混淆。使用ProGuard可以使得你的代码更加紧凑，这样能够使用更少mapped代码所需要的RAM。

14) 对最终的APK使用zipalign

在编写完所有代码，并通过编译系统生成APK之后，你需要使用zipalign对APK进行重新校准。如果你不做这个步骤，会导致你的APK需要更多的RAM，因为一些类似图片资源的东西不能被mapped。

> **Notes:** Google Play不接受没有经过zipalign的APK。

15) 分析你的RAM使用情况

一旦你获取到一个相对稳定的版本后，需要分析你的app整个生命周期内使用的内存情况，并进行优化，更多细节请参考Investigating Your RAM Usage.

16) 使用多进程

如果合适的话，有一个更高级的技术可以帮助你的app管理内存使用：通过把你的app组件切分成多个组件，运行在不同的进程中。这个技术必须谨慎使用，大多数app都不应该运行在多个进程中。因为如果使用不当，它会显著增加内存的使用，而不是减少。当你的app需要在后台运行与前台一样的大量的任务的时候，可以考虑使用这个技术。

一个典型的例子是创建一个可以长时间后台播放的Music Player。如果整个app运行在一个进程中，当后台播放的时候，前台的那些UI资源也没有办法得到释放。类似这样的app可以切分成2个进程：一个用来操作UI，另外一个用来后台的Service.

你可以通过在manifest文件中声明'android:process'属性来实现某个组件运行在另外一个进程的操作。

```
<service android:name=".PlaybackService"
         android:process=":background" />
```

更多关于使用这个技术的细节，请参考原文，链接如下。
http://developer.android.com/training/articles/memory.html

代码性能优化建议

> 编写:kesenhoo - 原文:http://developer.android.com/training/articles/perf-tips.html

这篇文章主要介绍一些小细节的优化技巧，虽然这些小技巧不能较大幅度的提升应用性能，但是恰当的运用这些小技巧并发生累积效应的时候，对于整个App的性能提升还是有不小作用的。通常来说，选择合适的算法与数据结构会是你首要考虑的因素，在这篇文章中不会涉及这方面的知识点。你应该使用这篇文章中的小技巧作为平时写代码的习惯，这样能够提升代码的效率。

通常来说，高效的代码需要满足下面两个原则：

- 不要做冗余的工作
- 尽量避免执行过多的内存分配操作

在优化App时其中一个难点就是让App能在各种型号的设备上运行。不同版本的虚拟机在不同的处理器上会有不同的运行速度。你甚至不能简单的认为“设备X的速度是设备Y的F倍”，然后还用这种倍数关系去推测其他设备。另外，在模拟器上的运行速度和在实际设备上的速度没有半点关系。同样，设备有没有JIT也对运行速度有重大影响：在有JIT情况下的最优化代码不一定在没有JIT的情况下也是最优的。

为了确保App在各设备上都能良好运行，就要确保你的代码在不同档次的设备上都尽可能的优化。

避免创建不必要的对象

创建对象从来不是免费的。**Generational GC**可以使临时对象的分配变得廉价一些，但是执行分配内存总是比不执行分配操作更昂贵。

随着你在App中分配更多的对象，你可能需要强制gc，而gc操作会给用户体验带来一点点卡顿。虽然从Android 2.3开始，引入了并发gc，它可以帮助你显著提升gc的效率，减轻卡顿，但毕竟不必要的内存分配操作还是应该尽量避免。

因此请尽量避免创建不必要的对象，有下面一些例子来说明这个问题：

- 如果你需要返回一个String对象，并且你知道它最终会需要连接到一个 `StringBuffer` ，请修改你的函数实现方式，避免直接进行连接操作，应该采用创建一个临时对象来做字符串的拼接这个操作。
- 当从已经存在的数据集中抽取出String的时候，尝试返回原数据的substring对象，而不是创建一个重复的对象。使用substring的方式，你将会得到一个新的String对象，但是这个string对象是和原string共享内部 `char[]` 空间的。

一个稍微激进点的做法是把所有多维的数据分解成一维的数组:

- 一组int数据要比一组Integer对象要好很多。可以得知，两组一维数组要比一个二维数组更加的有效率。同样的，这个道理可以推广至其他原始数据类型。
- 如果你需要实现一个数组用来存放(Foo,Bar)的对象，记住使用Foo[]与Bar[]要比(Foo,Bar)好很多。(例外的是，为了某些好的API的设计，可以适当做一些妥协。但是在自己的代码内部，你应该多多使用分解后的容易）。

通常来说，需要避免创建更多的临时对象。更少的对象意味者更少的gc动作，gc会对用户体验有比较直接的影响。

选择Static而不是Virtual

如果你不需要访问一个对象的值，请保证这个方法是static类型的，这样方法调用将快15%-20%。这是一个好的习惯，因为你可以从方法声明中得知调用无法改变这个对象的状态。

常量声明为Static Final

考虑下面这种声明的方式

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

编译器会使用一个初始化类的函数，然后当类第一次被使用的时候执行。这个函数将42存入 `intVal` ，还从class文件的常量表中提取了 `strVal` 的引用。当之后使用 `intVal` 或 `strVal` 的时候，他们会直接被查询到。

我们可以用 `final` 关键字来优化：

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

这时再也不需要上面的方法了，因为final声明的常量进入了静态dex文件的域初始化部分。调用 `intVal` 的代码会直接使用42，调用 `strVal` 的代码也会使用一个相对廉价的“字符串常量”指令，而不是查表。

> **Notes**：这个优化方法只对原始类型和String类型有效，而不是任意引用类型。不过，在必要时使用 `static final` 是个很好的习惯。

避免内部的Getters/Setters

像C++等native language，通常使用getters(i = getCount())而不是直接访问变量(i = mCount)。这是编写C++的一种优秀习惯，而且通常也被其他面向对象的语言所采用，例如C#与Java，因为编译器通常会做inline访问，而且你需要限制或者调试变量，你可以在任何时候在getter/setter里面添加代码。

然而，在Android上，这不是一个好的写法。虚函数的调用比起直接访问变量要耗费更多。在面向对象编程中，将getter和setting暴露给公用接口是合理的，但在类内部应该仅仅使用域直接访问。

在没有JIT(Just In Time Compiler)时，直接访问变量的速度是调用getter的3倍。有JIT时，直接访问变量的速度是通过getter访问的7倍。

请注意，如果你使用ProGuard，你可以获得同样的效果，因为ProGuard可以为你inline accessors.

使用增强的For循环

增强的For循环（也被称为 for-each 循环）可以被用在实现了 Iterable 接口的 collections 以及数组上。使用collection的时候，Iterator会被分配，用于for-each调用 `hasNext()` 和 `next()` 方法。使用ArrayList时，手写的计数式for循环会快3倍（不管有没有JIT），但是对于其他collection，增强的for-each循环写法会和迭代器写法的效率一样。

请比较下面三种循环的方法：

```
static class Foo {
    int mSplat;
}

Foo[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one() {
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;

    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two() {
    int sum = 0;
    for (Foo a : mArray) {
        sum += a.mSplat;
    }
}
```

- zero()是最慢的，因为JIT没有办法对它进行优化。
- one()稍微快些。
- two() 在没有做JIT时是最快的，可是如果经过JIT之后，与方法one()是差不多一样快的。它使用了增强的循环方法for-each。

所以请尽量使用for-each的方法，但是对于ArrayList，请使用方法one()。

> **Tips**：你还可以参考 Josh Bloch 的 《Effective Java》这本书的第46条

使用包级访问而不是内部类的私有访问

参考下面一段代码

```
public class Foo {
    private class Inner {
        void stuff() {
            Foo.this.doStuff(Foo.this.mValue);
        }
    }

    private int mValue;

    public void run() {
        Inner in = new Inner();
        mValue = 27;
        in.stuff();
    }

    private void doStuff(int value) {
        System.out.println("Value is " + value);
    }
}
```

这里重要的是，我们定义了一个私有的内部类（ `Foo$Inner` ），它直接访问了外部类中的私有方法以及私有成员对象。这是合法的，这段代码也会如同预期一样打印出"Value is 27"。

问题是，VM因为 `Foo` 和 `Foo$Inner` 是不同的类，会认为在 `Foo$Inner` 中直接访问 `Foo` 类的私有成员是不合法的。即使Java语言允许内部类访问外部类的私有成员。为了去除这种差异，编译器会产生一些仿造函数：

```
/*package*/ static int Foo.access$100(Foo foo) {
    return foo.mValue;
}
/*package*/ static void Foo.access$200(Foo foo, int value) {
    foo.doStuff(value);
}
```

每当内部类需要访问外部类中的mValue成员或需要调用doStuff()函数时，它都会调用这些静态方法。这意味着，上面的代码可以归结为，通过accessor函数来访问成员变量。早些时候我们说过，通过accessor会比直接访问域要慢。所以，这是一个特定语言用法造成性能降低的例子。

如果你正在性能热区（hotspot:高频率、重复执行的代码段）使用像这样的代码，你可以把内部类需要访问的域和方法声明为包级访问，而不是私有访问权限。不幸的是，这意味着在相同包中的其他类也可以直接访问这些域，所以在公开的API中你不能这样做。

避免使用float类型

Android系统中float类型的数据存取速度是int类型的一半，尽量优先采用int类型。

就速度而言，现代硬件上，float 和 double 的速度是一样的。空间而言，double 是两倍float的大小。在空间不是问题的情况下，你应该使用 double 。

同样，对于整型，有些处理器实现了硬件几倍的乘法，但是没有除法。这时，整型的除法和取余是在软件内部实现的，这在你使用哈希表或大量计算操作时要考虑到。

使用库函数

除了那些常见的让你多使用自带库函数的理由以外，记得系统函数有时可以替代第三方库，并且还有汇编级别的优化，他们通常比带有JIT的Java编译出来的代码更高效。典型的例子是：Android API 中的 `String.indexOf()`，Dalvik出于内联性能考虑将其替换。同样 `System.arraycopy()` 函数也被替换，这样的性能在Nexus One测试，比手写的for循环并使用JIT还快9倍。

> **Tips**：参见 Josh Bloch 的 《Effective Java》这本书的第47条

谨慎使用native函数

结合Android NDK使用native代码开发，并不总是比Java直接开发的效率更好的。Java转native代码是有代价的，而且JIT不能在这种情况下做优化。如果你在native代码中分配资源（比如native堆上的内存，文件描述符等等），这会对收集这些资源造成巨大的困难。你同时也需要为各种架构重新编译代码（而不是依赖JIT）。你甚至对已同样架构的设备都需要编译多个版本：为G1的ARM架构编译的版本不能完全使用Nexus One上ARM架构的优势，反之亦然。

Native 代码是在你已经有本地代码，想把它移植到Android平台时有优势，而不是为了优化已有的Android Java代码使用。

如果你要使用JNI,请学习JNI Tips

> **Tips**：参见 Josh Bloch 的 《Effective Java》这本书的第54条

关于性能的误区

在没有JIT的设备上，使用一种确切的数据类型确实要比抽象的数据类型速度要更有效率（例如，调用 `HashMap map` 要比调用 `Map map` 效率更高）。有误传效率要高一倍，实际上只是6%左右。而且，在JIT之后，他们直接并没有大多差异。

在没有JIT的设备上，读取缓存域比直接读取实际数据大概快20%。有JIT时，域读取和本地读取基本无差。所以优化并不值得除非你觉得能让你的代码更易读（这对 final, static, static final 域同样适用）。

关于测量

在优化之前，你应该确定你遇到了性能问题。你应该确保你能够准确测量出现在的性能，否则你也不会知道优化是否真的有效。

本章节中所有的技巧都需要Benchmark（基准测试）的支持。Benchmark可以在 code.google.com "dalvik" project 中找到

Benchmark是基于Java版本的 Caliper microbenchmarking框架开发的。Microbenchmarking很难做准确，所以Caliper帮你完成这部分工作，甚至还帮你测了你没想到需要测量的部分（因为，VM帮你管理了代码优化，你很难知道这部分优化有多大效果）。我们强烈推荐使用Caliper来做你的基准微测工作。

我们也可以用Traceview 来测量，但是测量的数据是没有经过JIT优化的，所以实际的效果应该是要比测量的数据稍微好些。

关于如何测量与调试，还可以参考下面两篇文章：

- Profiling with Traceview and dmtracedump
- Analysing Display and Performance with Systrace

提升Layout的性能

> 编写: allenlsy - 原文: http://developer.android.com/training/improving-layouts/index.html

Layout 是 Android 应用中直接影响用户体验的关键部分。如果实现的不好，你的 Layout 会导致程序非常占用内存并且 UI 运行缓慢。Android SDK 带有帮助你找到 Layout 性能问题的工具。结合本课内容使用它，你将学会使用最小的内存空间实现流畅的 UI。

Lessons

优化Layout的层级

就像一个复杂的网页会减慢载入速度，你的Layout结构如果太复杂，也会造成性能问题。本节教你如何使用SDK自带工具来查看Layout并找到性能瓶颈。

使用 `<include/>` 标签重用Layout

如果你的程序的 UI 在不同地方重复使用某个 Layout，那本节将教你如何创建高效的，可重用的Layout部件，并把它们“包含”到其他 UI Layout 中。

按需载入视图

除了简单的把一个 Layout 包含到另一个 Layout 中，你可能还想在程序开始之后，仅当你的 Layout 对用户可见时才开始载入。本节告诉你如何使用分步载入 Layout 来提高 Layout 的首次加载性能。

优化ListView的滑动性能

如果你有一个每个列表项 (item) 都包含很多数据或者复杂数据的 ListView ，那么列表滚动的性能很有可能会存在问题。本节会介绍给你一些如何优化滚动流畅度的技巧。

优化layout的层级

> 编写:[allenlsy](#) - 原文:http://developer.android.com/training/improving-layouts/optimizing-layout.html

一个常见的误区是，用最基础的 Layout 结构可以提高 Layout 的 性能。然而，因为程序的每个组件和 Layout 都需要经过初始化、布局和绘制的过程，如果布局嵌套导致层级过深，上面的初始化，布局和绘制操作就更加耗时。例如，使用嵌套的 LinearLayout 可能会使得 View 的层级结构过深，此外，嵌套使用了 `layout_weight` 参数的 LinearLayout 的计算量会尤其大，因为每个子元素都需要被测量两次。这对需要多次重复 inflate 的 Layout 尤其需要注意，比如嵌套在 ListView 或 GridView 时。

在本课中，你将学习使用 Hierarchy Viewer和Layoutopt来检查和优化 Layout。

检查 Layout

Android SDK 工具箱中有一个叫做 Hierarchy Viewer 的工具，能够在程序运行时分析 Layout。你可以用这个工具找到 Layout 的性能瓶颈。

Hierarchy Viewer 会让你选择设备或者模拟器上正在运行的进程，然后显示其 Layout 的树型结构。每个块上的交通灯分别代表了它在测量、布局和绘画时的性能，帮你找出瓶颈部分。

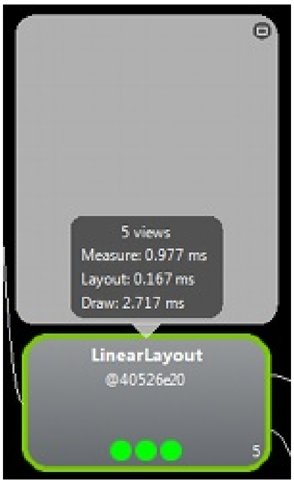
比如，下图是 ListView 中一个列表项的 Layout 。列表项里，左边放一个小位图，右边是两个层叠的文字。像这种需要被多次 inflate 的 Layout ，优化它们会有事半功倍的效果。



`hierarchyviewer` 这个工具在 `<sdk>/tools/` 中。当打开时，它显示一张可使用设备的列表，和它正在运行的组件。点击 **Load View Hierarchy** 来查看所选组件的层级。比如，下图就是前一个图中所示 Layout 的层级关系。



在上图中，你可以看到一个三层结构，其中右下角的 TextView 在布局的时候有问题。点击这个TextView可以看到每个步骤所花费的时间。

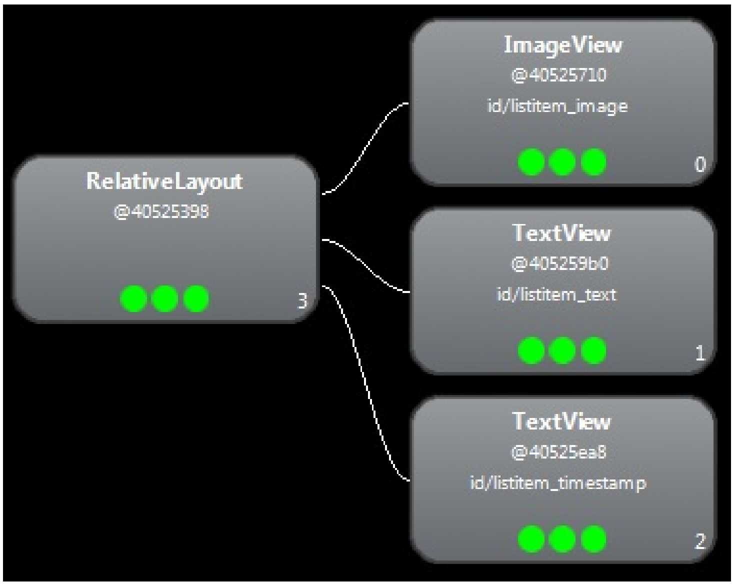


可以看到，渲染一个完整的列表项的时间就是：

- 测量: 0.977ms
- 布局: 0.167ms
- 绘制: 2.717ms

修正 Layout

上面的 Layout 由于有这个嵌套的 LinearLayout 导致性能太慢，可能的解决办法是将 Layout 层级扁平化 - 变浅变宽，而不是又窄又深。RelativeaLayout 作为根节点时就可以达到目的。所以，当换成基于 RelativeLayout 的设计时，你的 Layout 变成了两层。新的 Layout 变成这样：



现在渲染列表项的时间：

- 测量: 0.598ms
- 布局: 0.110ms
- 绘制: 2.146ms

可能看起来是很小的进步，但是由于它对列表中每个项都有效，这个时间要翻倍。

这个时间的主要差异是由于在 LinearLayout 中使用 `layout_weight` 所致，因为会减慢"测量"的速度。这只是一个正确使用各种 Layout 的例子，当你使用 `layout_weight` 时有必要慎重。

使用 Lint

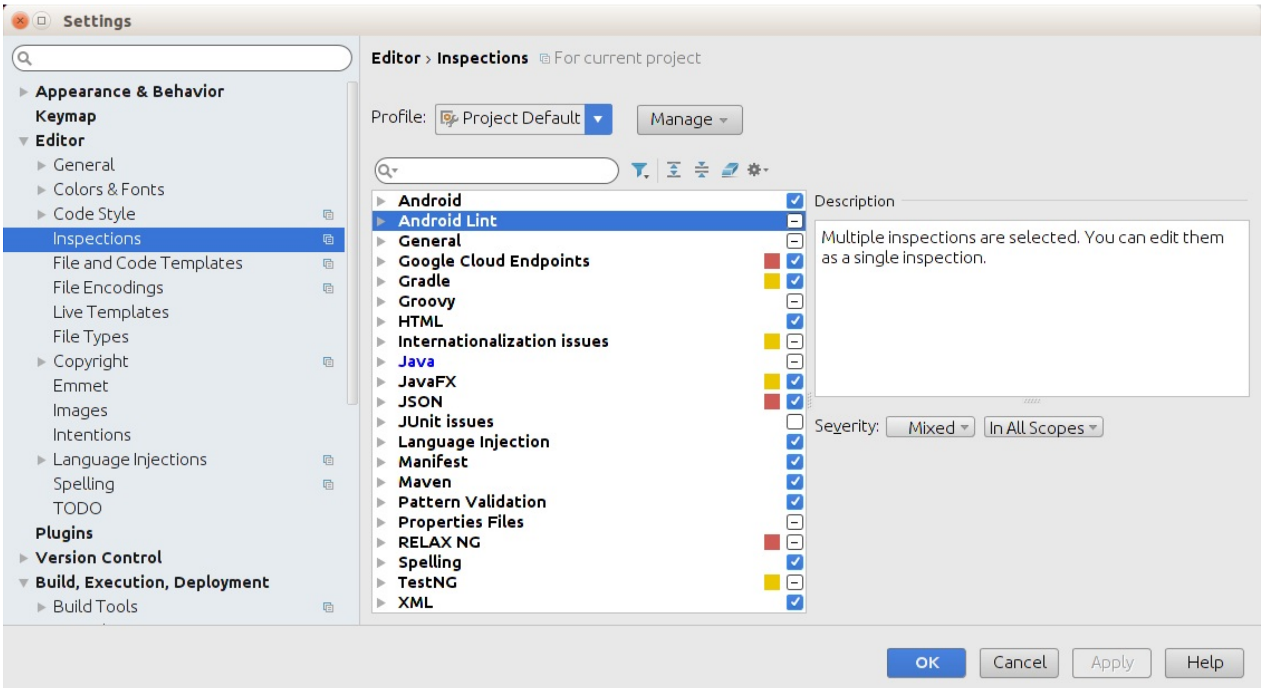
> 大部分叫做 lint 的编程工具，都是类似于代码规范的检测工具。比如JSLint，CSSLinkt，JSONLint 等等。译者注。

运行 Lint 工具来检查 Layout 可能的优化方法，是个很好的实践。Lint 已经取代了 Layoutopt 工具，它拥有更强大的功能。Lint 中包含的一些检测规则有：

- 使用compound drawable — 用一个compound drawable 替代一个包含 `ImageView` 和 `TextView` 的 `LinearLayout` 会更有效率。
- 合并根 frame — 如果 `FrameLayout` 是 Layout 的根节点，并且没有使用 padding 或者背景等，那么用 merge 标签替代他们会稍微高效些。
- 没用的子节点 — 一个没有子节点或者背景的 Layout 应该被去掉，来获得更扁平的层级
- 没用的父节点 — 一个节点如果没有兄弟节点，并且它不是 `ScrollView` 或根节点，没有背景，这样的节点应该直接被子节点取代，来获得更扁平的层级
- 太深的 Layout — Layout 的嵌套层数太深对性能有很大影响。尝试使用更扁平的 Layout，比如 `RelativeLayout` 或 `GridLayout` 来提高性能。一般最多不超过10层。

另一个使用 Lint 的好处就是，它内置于 Android Studio 中。Lint 在你导编译程序时自动运行。Android Studio 中，你可以为单独的 build variant 或者所有 variant 运行 lint。

你也可以在 Android Studio 中管理检测选项，在 **File > Settings > Project Settings** 中。检测配置页面会显示支持的检测项目。



Lint 有自动修复、提示建议和直接跳转到问题处的功能。

使用include标签重用layouts

> 编写:allenlsy - 原文:http://developer.android.com/training/improving-layouts/reusing-layouts.html

虽然 Android 提供很多小的可重用的交互组件，你仍然可能需要重用复杂一点的组件，这也许会用到 Layout。为了高效重用整个的 Layout，你可以使用 `<include/>` 和 `<merge/>` 标签把其他 Layout 嵌入当前 Layout。

重用 Layout 非常强大，它让你可以创建复杂的可重用 Layout。比如，一个 yes/no 按钮面板，或者带有文字的自定义进度条。这也意味着，任何在多个 Layout 中重复出现的元素可以被提取出来，被单独管理，再添加到 Layout 中。所以，虽然可以添加一个自定义 View 来实现单独的 UI 组件，你可以更简单的直接重用某个 Layout 文件。

创建可重用 Layout

如果你已经知道你需要重用的 Layout，就先创建一个新的 XML 文件并定义 Layout 。比如，以下是一个来自 G-Kenya codelab 的 Layout，定义了一个需要添加到每个 Activity 中的标题栏（titlebar.xml)：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width=”match_parent”
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</FrameLayout>
```

根节点 View 就是你想添加入的 Layout 类型。

使用 `<include>` 标签

使用 `<include>` 标签，可以在 Layout 中添加可重用的组件。比如，这里有一个来自 G-Kenya codelab 的 Layout 需要包含上面的那个标题栏：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width=”match_parent”
    android:layout_height=”match_parent”
    android:background="@color/app_bg"
    android:gravity="center_horizontal">

    <include layout="@layout/titlebar"/>

    <TextView android:layout_width=”match_parent”
              android:layout_height="wrap_content"
              android:text="@string/hello"
              android:padding="10dp" />

    ...

</LinearLayout>
```

你也可以覆写被添加的 Layout 的所有 Layout 参数（任何 android:layout_* 属性），通过在 `<include/>` 中声明他们来完成。比如：

```
<include android:id="@+id/news_title"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         layout="@layout/title"/>
```

然而，如果你要在 `<include>` 中覆写某些属性，你必须先覆写 `android:layout_height` 和 `android:layout_width` 。

使用 `<merge>` 标签

`<merge />` 标签在你嵌套 Layout 时取消了 UI 层级中冗余的 ViewGroup 。比如，如果你有一个 Layout 是一个竖直方向的 LinearLayout，其中包含两个连续的 View 可以在别的 Layout 中重用，那么你会做一个 LinearLayout 来包含这两个 View ，以便重用。不过，当使用一个 LinearLayout 作为另一个 LinearLayout 的根节点时，这种嵌套 LinearLayout 的方式除了减慢你的 UI 性能外没有任何意义。

为了避免这种情况，你可以用 `<merge>` 元素来替代可重用 Layout 的根节点。例如：

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/add"/>

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/delete"/>

</merge>
```

现在，当你要将这个 Layout 包含到另一个 Layout 中时（并且使用了 `<include/>` 标签），系统会忽略 `<merge>` 标签，直接把两个 Button 放到 Layout 中 `<include>` 的所在位置。

按需加载视图

> 编写:allenlsy - 原文:http://developer.android.com/training/improving-layouts/loading-ondemand.html

有时你的 Layout 会用到不怎么重用的复杂视图。不管它是列表项 细节，进度显示器，或是撤销时的提示信息，你可以仅在需要的时候载入它们，提高 UI 渲染速度。

定义 ViewStub

ViewStub 是一个轻量的视图，不需要大小信息，也不会在被加入的 Layout 中绘制任何东西。每个 ViewStub 只需要设置 `android:layout` 属性来指定需要被 inflate 的 Layout 类型。

以下 ViewStub 是一个半透明的进度条覆盖层。功能上讲，它应该只在新的数据项被导入到应用程序时可见。

```
<ViewStub
    android:id="@+id/stub_import"
    android:inflatedId="@+id/panel_import"
    android:layout="@layout/progress_overlay"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom" />
```

载入 ViewStub Layout

当你要载入用 ViewStub 声明的 Layout 时，要么用 `setVisibility(View.VISIBLE)` 设置它的可见性，要么调用其 `inflate()` 方法。

```
((ViewStub) findViewById(R.id.stub_import)).setVisibility(View.VISIBLE);
// or
View importPanel = ((ViewStub) findViewById(R.id.stub_import)).inflate();
```

> **Notes**：`inflate()` 方法会在渲染完成后返回被 inflate 的视图，所以如果你需要和这个 Layout 交互的话， 你不需要再调用 `findViewById()` 去查找这个元素，。

一旦 ViewStub 可见或是被 inflate 了，ViewStub 就不再继续存在View的层级机构中了。取而代之的是被 inflate 的 Layout，其 id 是 ViewStub 上的 `android:inflatedId` 属性。（ViewStub 的 `android:id` 属性仅在 ViewStub 可见以前可用）

> **Notes**：ViewStub 的一个缺陷是，它目前不支持使用 `<merge/>` 标签的 Layout 。

使得ListView滑动顺畅

> 编写:allenlsy - 原文:http://developer.android.com/training/improving-layouts/smooth-scrolling.html

保持程序流畅的关键，是让主线程（UI 线程）不要进行大量运算。你要确保在其他线程执行磁盘读写、网络读写或是 SQL 操作等。为了测试你的应用的状态，你可以启用 StrictMode。

使用后台线程

你应该把主线程中的耗时间的操作，提取到一个后台线程（也叫做“worker thread工作线程”）中，使得主线程只关注 UI 绘画。很多时候，使用 AsyncTask 是一个简单的在主线程以外进行操作的方法。系统会自动把 `execute()` 的请求放入队列中并线性调用执行。这个行为是全局的，这意味着你不需要考虑自己定义线程池的事情。

在下面的例子中，一个 AsyncTask 被用于在后台线程载入图片，并在载入完成后把图片显示到 UI 上。当图片正在载入时，它还会显示一个进度提示。

```
// Using an AsyncTask to load the slow images in a background thread
new AsyncTask<ViewHolder, Void, Bitmap>() {
    private ViewHolder v;

    @Override
    protected Bitmap doInBackground(ViewHolder... params) {
        v = params[0];
        return mFakeImageLoader.getImage();
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        super.onPostExecute(result);
        if (v.position == position) {
            // If this item hasn't been recycled already, hide the
            // progress and set and show the image
            v.progress.setVisibility(View.GONE);
            v.icon.setVisibility(View.VISIBLE);
            v.icon.setImageBitmap(result);
        }
    }
}.execute(holder);
```

从 Android 3.0 (API level 11) 开始, AsyncTask 有个新特性，那就是它可以在多个 CPU 核上运行。你可以调用 `executeOnExecutor()` 而不是 `execute()` ，前者可以根据CPU的核心数来触发多个任务同时进行。

在 ViewHolder 中填入视图对象

你的代码可能在 ListView 滑动时经常使用 `findViewById()` ，这样会降低性能。即使是 Adapter 返回一个用于回收的 inflate 后的视图，你仍然需要查看这个元素并更新它。避免频繁调用 `findViewById()` 的方法之一，就是使用 ViewHolder（视图占位符）的设计模式。

一个 ViewHolder 对象存储了他的标签下的每个视图。这样你不用频繁查找这个元素。第一，你需要创建一个类来存储你会用到的视图。比如：

```
static class ViewHolder {
  TextView text;
  TextView timestamp;
  ImageView icon;
  ProgressBar progress;
  int position;
}
```

然后，在 Layout 的类中生成一个 ViewHolder 对象：

```
ViewHolder holder = new ViewHolder();
holder.icon = (ImageView) convertView.findViewById(R.id.listitem_image);
holder.text = (TextView) convertView.findViewById(R.id.listitem_text);
holder.timestamp = (TextView) convertView.findViewById(R.id.listitem_timestamp);
holder.progress = (ProgressBar) convertView.findViewById(R.id.progress_spinner);
convertView.setTag(holder);
```

这样你就可以轻松获取每个视图，而不是使用 `findViewById()` 来不断查找子视图，节省了宝贵的运算时间。

优化电池寿命

> 编写:kesenhoo - 原文:http://developer.android.com/training/monitoring-device-state/index.html

显然，手持设备的电量使用情况需要引起很大的重视。通过这一系列的课程，你将学会如何根据设备的状态来改变App的某些行为与功能。

通过在失去网络连接时关闭后台更新服务，在剩余电量较低时减少更新数据的频率等操作，你可以在不影响用户体验的前提下，确保App对电池寿命的影响减到最小。

课程

检测电量与充电状态

学习如何通过判断与检测当前电池电量以及充电状态的变化，改变应用程序的更新频率。

判断并监测设备的底座状态与类型

设备使用习惯的区别也会影响到刷新频率的优化措施，这节课中将学习如何判断与监测底座状态及其种类来改变应用程序的行为。

判断并检测网络连接状态

在没有连接到互联网的情况下，你是无法在线更新应用的。这一节课将学习如何根据网络的连接状态，改变后台更新的频率，以及如何在高带宽传输任务开始前，判断网络连接类型(Wi-Fi/数据连接)。

按需操纵BroadcastReceiver

在Manifest清单文件中声明的BroadcastReceiver可以在运行时切换其开启状态，这样一来，我们就可以根据当前设备的状态，禁用那些没有必要开启的BroadcastReceiver。在这一节课将学习如何通过切换这些BroadcastReceiver的开启状态，以及如何根据设备的状态延迟某一操作的执行时机，来提高应用的效率。

监测电池的电量与充电状态

> 编写:kesenhoo - 原文:http://developer.android.com/training/monitoring-device-state/battery-monitoring.html

当你想通过改变后台更新操作的频率来减少对电池寿命的影响时，那么首先需要检查当前电量与充电状态。

执行应用更新对电池寿命的影响是与电量和充电状态密切相关的。当使用交流电对设备充电时，更新操作的影响可以忽略不计，所以在大多数情况下，如果使用壁式充电器对设备进行充电，我们可以将刷新频率设置到最大。相反的，如果设备没有在充电状态，那么我们就需要尽量减少设备的更新操作来延长电池的续航能力。

同样的，如果我们监测到电量即将耗尽时，那么应该尽可能降低甚至停止更新操作。

判断当前充电状态

首先来看一下应该如何确定当前的充电状态。BatteryManager会广播一个带有电池与充电详情的Sticky Intent

因为广播的是一个sticky Intent，所以不需要注册BroadcastReceiver。仅仅只需要调用一个以 `null` 作为Receiver参数的 `registerReceiver()` 方法就可以了。如下面的代码片段中展示的那样，它返回了保存当前电池信息的Intent。你也可以在这里传入一个实际的BroadcastReceiver对象，但这并不是必须的。

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
```

我们可以提取出当前的充电状态，以及设备处于充电时，是通过USB还是交流充电器充电的。

```
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

通常，我们可以在设备使用交流充电时最大化后台更新频率，在使用USB充电时降低更新频率，在非充电状态时，将更新频率进一步降低。

监测充电状态的改变

充电状态随时可能改变，所以我们应该检查充电状态的改变来调整更新频率。

BatteryManager会在设备连接或者断开充电器的时候广播一个Action。即使应用没有运行，我们也应该接收这些事件的广播，主要原因是因为这些事件会影响到应用启动（从而进行更新）的频率，因此我们应该在Manifest文件里面注册一个BroadcastReceiver来监听含有ACTION_POWER_CONNECTED 与 ACTION_POWER_DISCONNECTED的Intent。

```
<receiver android:name=".PowerConnectionReceiver">
  <intent-filter>
    <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
    <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
  </intent-filter>
</receiver>
```

我们可以在该BroadcastReceiver的实现中，提取出当前的充电状态，如下所示：

```
public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
        boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                            status == BatteryManager.BATTERY_STATUS_FULL;

        int chargePlug = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
        boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
    }
}
```

判断当前电池电量

在一些情况下，获取到当前电池电量也很有帮助。我们可以在获知电量少于某个级别的时候减少后台的更新频率。 我们可以通过电池状态Intent获取到电池电量与容量等信息，如下所示：

```
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

float batteryPct = level / (float)scale;
```

检测电量的有效改变

我们不能不停地监测电池状态，实际上这也是不必要的。通常来说，不间断地监测电量信息对电池的影响会远大于应用本身对电池的影响。所以我们应该仅监测电量的一些显著性变化，特别是当设备进入或者离开低电量状态时。

在下面的Manifest文件片段中，BroadcastReceiver仅仅监听 `ACTION_BATTERY_LOW` 与 `ACTION_BATTERY_OKAY` ，这样它就只会在设备电量进入低电量或者离开低电量的时候被触发。

```
<receiver android:name=".BatteryLevelReceiver">
<intent-filter>
  <action android:name="android.intent.action.ACTION_BATTERY_LOW"/>
  <action android:name="android.intent.action.ACTION_BATTERY_OKAY"/>
  </intent-filter>
</receiver>
```

通常我们都需要在进入低电量的情况下，关闭所有后台更新来维持设备的续航，因为这个时候做任何更新等操作都极有可能是无用的，因为也许在你还没来得及处理更新的数据时，设备就因电量耗尽而自动关机了。

在很多时候，用户往往会将设备放入某种底座中充电（译注：比如车载的底座式充电器），在下一节课程当中，我们将会学习如何确定当前的底座状态，以及如何监听设备底座的变化。

判断并监测设备的底座状态与类型

> 编写:kesenhoo - 原文:http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html

Android设备可以放置在许多不同的底座中，包括车载底座，家庭底座还有数字信号底座以及模拟信号底座等。由于许多底座会向设备充电，因此底座状态通常与充电状态密切相关。

你的应用类型决定了底座类型会对更新频率产生怎样的影响。对于一个体育类应用，可以让设备在笔记本底座状态下增加更新的频率，或者当设备在车载底座状态下停止更新。相反的，如果你的后台服务用来更新交通数据，你也可以选择在车载底座模式下最大化更新的频率。

底座状态也是以Sticky Intent方式来广播的，这样可以通过查询Intent里面的数据来判断目前设备是否放置在底座中，以及底座的类型。

判断当前底座状态

底座状态的具体信息会以Extra数据的形式，包含在具有ACTION_DOCK_EVENT这一Action的某个Sticky广播中 ，因此，你不需要为其注册一个BroadcastReceiver。如下所示，仅需要将 `null` 作为参数传递给registerReceiver()方法就可以了：

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_DOCK_EVENT);
Intent dockStatus = context.registerReceiver(null, ifilter);
```

你可以从 `EXTRA_DOCK_STATE` 这一Extra数据中，提取出当前的底座状态：

```
int dockState = battery.getIntExtra(EXTRA_DOCK_STATE, -1);
boolean isDocked = dockState != Intent.EXTRA_DOCK_STATE_UNDOCKED;
```

判断当前底座类型

如果设备被放置在了底座中，那么它可以有下面四种底座类型：

- Car
- Desk
- Low-End (Analog) Desk
- High-End (Digital) Desk

注意最后两种底座类型仅在API Level 11及以后版本的Android系统中才被支持。如果你只在乎底座的类型而不管它是数字的还是模拟的，那么可以仅监测三种类型：

```
boolean isCar = dockState == EXTRA_DOCK_STATE_CAR;
boolean isDesk = dockState == EXTRA_DOCK_STATE_DESK ||
                 dockState == EXTRA_DOCK_STATE_LE_DESK ||
                 dockState == EXTRA_DOCK_STATE_HE_DESK;
```

监测底座状态或者类型的改变

当设备被放置在或者拔出底座时，系统会发出一个具有ACTION_DOCK_EVENT这一Action的广播。为了监听底座状态的变化，我们只需要在应用的Manifest文件中注册一个BroadcastReceiver，如下所示：

```
<action android:name="android.intent.action.ACTION_DOCK_EVENT"/>
```

之于该BroadcastReceiver的具体实现，可以参考前面提到的那些方法，以此来提取出当前的底座类型和状态。

判断并监测网络连接状态

> 编写:kesenhoo - 原文:http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html

重复闹钟和后台服务最常见的功能之一，是用来从网络上获取应用更新，存储数据或者执行大文件的下载。但是如果没有获得网络连接，或者连接的速度太慢以至于下载无法完成，那么就没有必要唤醒设备并执行那些更新等操作了。

我们可以使用ConnectivityManager来检查设备是否连接到网络，以及网络的类型（译注：通过网络的连接状况改变，相应的改变app的行为，减少无谓的操作，从而延长设备的续航能力）。

判断当前是否有网络连接

如果没有网络连接，那么就没有必要做那些需要联网的事情。下面的代码片段展示了如何通过ConnectivityManager检查当前活动的网络类型，并确定它是否可以连接到互联网：

```
ConnectivityManager cm =
        (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected = activeNetwork != null &&
                      activeNetwork.isConnectedOrConnecting();
```

判断连接网络的类型

我们还可以获取到当前的网络连接类型。

设备通常可以有移动网络，WiMax，Wi-Fi与以太网连接等类型。通过查询当前活动的网络类型，可以根据网络的带宽对更新频率进行调整：

```
boolean isWiFi = activeNetwork.getType() == ConnectivityManager.TYPE_WIFI;
```

移动网络的使用费会比Wi-Fi更高，所以多数情况下，如果设备正在使用移动网络，我们应该减少应用的更新频率；同样地，还应该临时地挂起一些文件下载任务直到有Wi-Fi连接时再继续下载。

如果已经关闭了更新操作，那么需要监听网络连接的变化，这样就可以在建立了互联网访问之后，重新恢复它们。

监听网络连接的变化

当网络连接发生改变时，ConnectivityManager会广播CONNECTIVITY_ACTION（`android.net.conn.CONNECTIVITY_CHANGE`）的Action消息。我们可以在Manifest文件里面注册一个BroadcastReceiver，来监听这些变化，并适当地恢复（或挂起）你的后台更新:

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

设备的网络变化可能会比较频繁，因此每当你在移动网络与Wi-Fi之间切换的时候，这一广播就会被触发。因此，我们可以仅在之前的更新或者下载任务被挂起的时候去监听这一广播（用来恢复那些任务）。通常我们可以在开始更新前检查一下网络连接，如果当前没有连接到互联网，那么就将更新任务挂起，直到连接恢复。

上述方法会涉及到Broadcast Receiver开启状态的切换，这一内容会在下一节课中展开。

按需操控BroadcastReceiver

> 编写:kesenhoo - 原文:http://developer.android.com/training/monitoring-device-state/manifest-receivers.html

监测设备状态变化最简单的方法，是为你所要监听的每一个状态创建一个BroadcastReceiver，并在Manifest文件中注册它们。之后就可以在每一个BroadcastReceiver中，根据当前设备的状态调整一些计划任务。

上述方法的副作用是：一旦你的接收器收到了广播，应用就会唤醒设备。唤醒的频率可能会远高于需要的频率

更好的方法是在程序运行时开启或者关闭BroadcastReceiver。这样的话，你就可以让这些接收器仅在需要的时候被激活。

切换是否开启接收器以提高效率

我们可以使用PackageManager来切换任何一个在Mainfest里面定义好的组件的开启状态。通过下面的方法可以开启或者关闭任何一个BroadcastReceiver：

```
ComponentName receiver = new ComponentName(context, myReceiver.class);

PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP)
```

使用这种技术，如果我们确定网络连接已经断开，那么可以在这个时候关闭除了监听网络状态变化的接收器之外的其它所有接收器。

相反的，一旦重新建立网络连接，我们可以停止监听网络连接的改变，而仅仅在执行需要联网的操作之前判断当前网络是否可以用。

同样地，你可以使用上面的技术来暂缓一个需要更高带宽的下载任务。这仅需要启用一个监听网络连接变化的BroadcastReceiver，并在连接到Wi-Fi时，初始化下载任务。

多线程操作

> 编写:AllenZheng1991 - 原文:http://developer.android.com/training/multiple-threads/index.html

把一个相对耗时且数据操作复杂的任务分割成多个小的操作，然后分别运行在多个线程上，这能够提高完成任务的速度和效率。在多核CPU的设备上，系统可以并行运行多个线程，而不需要让每个子操作等待CPU的时间片切换。例如，如果要解码大量的图片文件并以缩略图的形式把图片显示在屏幕上，当你把每个解码操作单独用一个线程去执行时，会发现速度快了很多。

这个章节会向你展示如何在一个Android应用中创建和使用多线程，以及如何使用线程池对象（thread pool object）。你还将了解到如何使得代码运行在指定的线程中，以及如何让你创建的线程和UI线程进行通信。

Sample Code

点击下载：**ThreadSample**

Lessons

在一个线程中执行一段特定的代码

学习如何通过实现Runnable接口定义一个线程类，让你写的代码能在单独的一个线程中执行。

为多线程创建线程池

学习如何创建一个能管理线程池和任务队列的对象，需要使用一个叫ThreadPoolExecutor的类。

在线程池中的一个线程里执行代码

学习如何让线程池里的一个线程执行一个任务。

与UI线程通信

学习如何让线程池里的一个普通线程与UI线程进行通信。

在一个线程中执行一段特定的代码

> 编写:AllenZheng1991 - 原文:http://developer.android.com/training/multiple-threads/define-runnable.html

这一课向你展示了如何通过实现 Runnable接口得到一个能在重写的 `Runnable.run()` 方法中执行一段代码的单独的线程。另外你可以传递一个Runnable对象到另一个对象，然后这个对象可以把它附加到一个线程，并执行它。一个或多个执行特定操作的Runnable对象有时也被称为一个任务。

Thread和Runnable只是两个基本的线程类，通过他们能发挥的作用有限，但是他们是强大的Android线程类的基础类，例如Android中的HandlerThread, AsyncTask和IntentService都是以它们为基础。Thread和Runnable同时也是ThreadPoolExecutor类的基础。ThreadPoolExecutor类能自动管理线程和任务队列，甚至可以并行执行多个线程。

定义一个实现Runnable接口的类

直接了当的方法是通过实现Runnable接口去定义一个线程类。例如：

```
public class PhotoDecodeRunnable implements Runnable {
    ...
    @Override
    public void run() {
        /*
         * 把你想要在线程中执行的代码写在这里
         */
        ...
    }
    ...
}
```

实现run()方法

在一个类里，`Runnable.run()` 包含执行了的代码。通常在Runnable 中执行任何操作都是可以的，但需要记住的是，因为Runnable 不会在UI线程中运行，所以它不能直接更新UI对象，例如View 对象。为了与UI对象进行通信，你必须使用另一项技术，在与UI线程进行通信 这一课中我们会对其进行描述。

在Runnable.run())方法的开始的地方通过调用参数为THREAD_PRIORITY_BACKGROUND的Process.setThreadPriority()方法来设置线程使用的是后台运行优先级。 这个方法减少了通过Runnable创建的线程和和UI线程之间的资源竞争。

你还应该通过在**Runnable** 自身中调用**Thread.currentThread()**来存储一个引用到**Runnable**对象的线程。

下面这段代码展示了如何创建run()方法：

```
class PhotoDecodeRunnable implements Runnable {
...
    /*
     * 定义要在这个任务中执行的代码
     */
    @Override
    public void run() {
        // 把当前的线程变成后台执行的线程
        android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGR
OUND);
        ...
        /*
         * 在PhotoTask实例中存储当前线程，以至于这个实例能中断这个线程
         */
        mPhotoTask.setImageDecodeThread(Thread.currentThread());
        ...
    }
...
}
```

为多线程创建管理器

> 编写:AllenZheng1991 - 原文:http://developer.android.com/training/multiple-threads/create-threadpool.html

在前面的课程中展示了如何在单独的一个线程中执行一个任务。如果你的线程只想执行一次，那么上一课的内容已经能满足你的需要了。

如果你想在一个数据集中重复执行一个任务，而且你只需要一个执行运行一次。这时，使用一个IntentService将能满足你的需求。 为了在资源可用的的时候自动执行任务，或者允许不同的任务同时执行（或前后两者），你需要提供一个管理线程的集合。 为了做这个管理线程的集合，使用一个ThreadPoolExecutor实例，当一个线程在它的线程池中变得不受约束时，它会运行队列中的一个任务。 为了能执行这个任务，你所需要做的就是把它加入到这个队列。

一个线程池能运行多个并行的任务实例，因此你要能保证你的代码是线程安全的，从而你需要给会被多个线程访问的变量附上同步代码块(synchronized block)。 当一个线程在对一个变量进行写操作时，通过这个方法将能阻止另一个线程对该变量进行读取操作。 典型的，这种情况会发生在静态变量上，但同样它也能突然发生在任意一个只实例化一次。 为了学到更多的相关知识，你可以阅读进程与线程这一API指南。

定义线程池类

在自己的类中实例化ThreadPoolExecutor类。在这个类里需要做以下事：

1. 为线程池使用静态变量

为了有一个单一控制点用来限制CPU或涉及网络资源的Runnable类型，你可能需要有一个能管理所有线程的线程池，且每个线程都会是单个实例。比如，你可以把这个作为一部分添加到你的全局变量的声明中去：

```
public class PhotoManager {
    ...
    static  {
        ...
        // Creates a single static instance of PhotoManager
        sInstance = new PhotoManager();
    }
    ...
```

2. 使用私有构造方法

让构造方法私有从而保证这是一个单例，这意味着你不需要在同步代码块(synchronized block)中额外访问这个类：

```
public class PhotoManager {
    ...
    /**
     * Constructs the work queues and thread pools used to download
     * and decode images. Because the constructor is marked private,
     * it's unavailable to other classes, even in the same package.
     */
    private PhotoManager() {
    ...
    }
```

3.通过调用线程池类里的方法开启你的任务

在线程池类中定义一个能添加任务到线程池队列的方法。例如：

```
public class PhotoManager {
    ...
    // Called by the PhotoView to get a photo
    static public PhotoTask startDownload(
        PhotoView imageView,
        boolean cacheFlag) {
        ...
        // Adds a download task to the thread pool for execution
        sInstance.
                mDownloadThreadPool.
                execute(downloadTask.getHTTPDownloadRunnable());
        ...
    }
```

4. 在构造方法中实例化一个**Handler**，且将它附加到你**APP**的**UI**线程。

一个Handler允许你的APP安全地调用UI对象（例如 View对象）的方法。大多数UI对象只能从UI线程安全的代码中被修改。这个方法将会在与UI线程进行通信(Communicate with the UI Thread)这一课中进行详细的描述。例如：

```
private PhotoManager() {
    ...
        // Defines a Handler object that's attached to the UI thread
        mHandler = new Handler(Looper.getMainLooper()) {
            /*
             * handleMessage() defines the operations to perform when
             * the Handler receives a new Message to process.
             */
            @Override
            public void handleMessage(Message inputMessage) {
                ...
            }
        ...
        }
    }
```

确定线程池的参数

一旦有了整体的类结构,你可以开始定义线程池了。为了初始化一个ThreadPoolExecutor对象，你需要提供以下数值：

1. 线程池的初始化大小和最大的大小

这个是指最初分配给线程池的线程数量，以及线程池中允许的最大线程数量。在线程池中拥有的线程数量主要取决于你的设备的CPU内核数。

这个数字可以从系统环境中获得：

```
public class PhotoManager {
...
    /*
     * Gets the number of available cores
     * (not always the same as the maximum number of cores)
     */
    private static int NUMBER_OF_CORES =
            Runtime.getRuntime().availableProcessors();
}
```

这个数字可能并不反映设备的物理核心数量，因为一些设备根据系统负载关闭了一个或多个CPU内核，对于这样的设备， `availableProcessors()` 方法返回的是处于活动状态的内核数量，可能少于设备的实际内核总数。

2.线程保持活动状态的持续时间和时间单位

这个是指线程被关闭前保持空闲状态的持续时间。这个持续时间通过时间单位值进行解译，是TimeUnit()中定义的常量之一。

3.一个任务队列

这个传入的队列由ThreadPoolExecutor获取的Runnable对象组成。为了执行一个线程中的代码，一个线程池管理者从先进先出的队列中取出一个Runnable对象且把它附加到一个线程。当你创建线程池时需要提供一个队列对象，这个队列对象类必须实现BlockingQueue接口。为了满足你的APP的需求，你可以选择一个Android SDK中已经存在的队列实现类。为了学习更多相关的知识，你可以看一下ThreadPoolExecutor类的概述。下面是一个使用LinkedBlockingQueue实现的例子：

```
public class PhotoManager {
    ...
    private PhotoManager() {
        ...
        // A queue of Runnables
        private final BlockingQueue<Runnable> mDecodeWorkQueue;
        ...
        // Instantiates the queue of Runnables as a LinkedBlockingQueue
        mDecodeWorkQueue = new LinkedBlockingQueue<Runnable>();
        ...
    }
    ...
}
```

创建一个线程池

为了创建一个线程池，可以通过调用ThreadPoolExecutor()构造方法初始化一个线程池管理者对象，这样就能创建和管理一组可约束的线程了。如果线程池的初始化大小和最大大小相同，ThreadPoolExecutor在实例化的时候就会创建所有的线程对象。例如：

```
private PhotoManager() {
        ...
        // Sets the amount of time an idle thread waits before terminating
        private static final int KEEP_ALIVE_TIME = 1;
        // Sets the Time Unit to seconds
        private static final TimeUnit KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;
        // Creates a thread pool manager
        mDecodeThreadPool = new ThreadPoolExecutor(
                NUMBER_OF_CORES,       // Initial pool size
                NUMBER_OF_CORES,       // Max pool size
                KEEP_ALIVE_TIME,
                KEEP_ALIVE_TIME_UNIT,
                mDecodeWorkQueue);
    }
```

启动与停止线程池中的线程

> 编写:AllenZheng1991 - 原文:http://developer.android.com/training/multiple-threads/run-code.html

在前面的课程中向你展示了如何去定义一个可以管理线程池且能在他们中执行任务代码的类。在这一课中我们将向你展示如何在线程池中执行任务代码。为了达到这个目的，你需要把任务添加到线程池的工作队列中去，当一个线程变成可运行状态时，ThreadPoolExecutor从工作队列中取出一个任务，然后在该线程中执行。

这节课同时也向你展示了如何去停止一个正在执行的任务，这个任务可能在刚开始执行时是你想要的，但后来发现它所做的工作并不是你所需要的。你可以取消线程正在执行的任务，而不是浪费处理器的运行时间。例如你正在从网络上下载图片且对下载的图片进行了缓存，当检测到正在下载的图片在缓存中已经存在时，你可能希望停止这个下载任务。当然，这取决于你编写APP的方式，因为可能压在你启动下载任务之前无法获知是否需要启动这个任务。

启动线程池中的线程执行任务

为了在一个特定的线程池的线程里开启一个任务，可以通过调用ThreadPoolExecutor.execute()，它需要提供一个Runnable类型的参数，这个调用会把该任务添加到这个线程池中的工作队列。当一个空闲的线程进入可执行状态时，线程管理者从工作队列中取出等待时间最长的那个任务，并且在线程中执行它。

```
public class PhotoManager {
    public void handleState(PhotoTask photoTask, int state) {
        switch (state) {
            // The task finished downloading the image
            case DOWNLOAD_COMPLETE:
            // Decodes the image
                mDecodeThreadPool.execute(
                        photoTask.getPhotoDecodeRunnable());
            ...
        }
        ...
    }
    ...
}
```

当ThreadPoolExecutor在一个线程中开启一个Runnable后，它会自动调用Runnable的run()方法。

中断正在执行的代码

为了停止执行一个任务，你必须中断执行这个任务的线程。在准备做这件事之前，当你创建一个任务时，你需要存储处理该任务的线程。例如：

```
class PhotoDecodeRunnable implements Runnable {
    // Defines the code to run for this task
    public void run() {
        /*
         * Stores the current Thread in the
         * object that contains PhotoDecodeRunnable
         */
        mPhotoTask.setImageDecodeThread(Thread.currentThread());
        ...
    }
    ...
}
```

想要中断一个线程，你可以调用Thread.interrupt()。需要注意的是这些线程对象都被系统控制，系统可以在你的APP进程之外修改 他们。因为这个原因，在你要中断一个线程时，你需要把这段代码放在一个同步代码块中对这个线程的访问加锁来解决这个问题。例如：

```
public class PhotoManager {
    public static void cancelAll() {
        /*
         * Creates an array of Runnables that's the same size as the
         * thread pool work queue
         */
        Runnable[] runnableArray = new Runnable[mDecodeWorkQueue.size()];
        // Populates the array with the Runnables in the queue
        mDecodeWorkQueue.toArray(runnableArray);
        // Stores the array length in order to iterate over the array
        int len = runnableArray.length;
        /*
         * Iterates over the array of Runnables and interrupts each one's Thread.
         */
        synchronized (sInstance) {
            // Iterates over the array of tasks
            for (int runnableIndex = 0; runnableIndex < len; runnableIndex++) {
                // Gets the current thread
                Thread thread = runnableArray[taskArrayIndex].mThread;
                // if the Thread exists, post an interrupt to it
                if (null != thread) {
                    thread.interrupt();
                }
            }
        }
    }
    ...
}
```

在大多数情况下，通过调用Thread.interrupt()能立即中断这个线程，然而他只能停止那些处于等待状态的线程，却不能中断那些占据CPU或者耗时的连接网络的任务。为了避免拖慢系统速度或造成系统死锁，在尝试执行耗时操作之前，你应该测试当前是否存在处于挂起状态的中断请求：

```
/*
 * Before continuing, checks to see that the Thread hasn't
 * been interrupted
 */
if (Thread.interrupted()) {
    return;
}
...
// Decodes a byte array into a Bitmap (CPU-intensive)
BitmapFactory.decodeByteArray(
        imageBuffer, 0, imageBuffer.length, bitmapOptions);
...
```

与UI线程通信

> 编写:AllenZheng1991 - 原文:http://developer.android.com/training/multiple-threads/communicate-ui.html

在前面的课程中你学习了如何在一个被ThreadPoolExecutor管理的线程中开启一个任务。最后这一节课将会向你展示如何从执行的任务中发送数据给运行在UI线程中的对象。这个功能允许你的任务可以做后台工作，然后把得到的结果数据转移给UI元素使用，例如位图数据。

任何一个APP都有自己特定的一个线程用来运行UI对象，比如View对象，这个线程我们称之为UI线程。只有运行在UI线程中的对象能访问运行在其它线程中的对象。因为你的任务执行的线程来自一个线程池而不是执行在UI线程，所以他们不能访问UI对象。为了把数据从一个后台线程转移到UI线程，需要使用一个运行在UI线程里的Handler。

在UI线程中定义一个Handler

Handler属于Android系统的线程管理框架的一部分。一个Handler对象用于接收消息和执行处理消息的代码。一般情况下，如果你为一个新线程创建了一个Handler，你还需要创建一个Handler，让它与一个已经存在的线程关联，用于这两个线程之间的通信。如果你把一个Handler关联到UI线程，处理消息的代码就会在UI线程中执行。

你可以在一个用于创建你的线程池的类的构造方法中实例化一个Handler对象，并把它定义为全局变量，然后通过使用Handler (Looper) 这一构造方法实例化它，用于关联到UI线程。Handler(Looper)这一构造方法需要传入了一个Looper对象，它是Android系统的线程管理框架中的另一部分。当你在一个特定的Looper实例的基础上去实例化一个Handler时，这个Handler与Looper运行在同一个线程里。例如：

```
private PhotoManager() {
...
    // Defines a Handler object that's attached to the UI thread
    mHandler = new Handler(Looper.getMainLooper()) {
    ...
```

在这个Handler里需要重写handleMessage()方法。当这个Handler接收到由另外一个线程管理的Handler发送过来的新消息时，Android系统会自动调用这个方法，而所有线程对应的Handler都会收到相同信息。例如：

```
        /*
         * handleMessage() defines the operations to perform when
         * the Handler receives a new Message to process.
         */
        @Override
        public void handleMessage(Message inputMessage) {
            // Gets the image task from the incoming Message object.
            PhotoTask photoTask = (PhotoTask) inputMessage.obj;
            ...
        }
    ...
    }
}
```

下一部分将向你展示如何用Handler转移数据。

把数据从一个任务中转移到UI线程

为了从一个运行在后台线程的任务对象中转移数据到UI线程中的一个对象，首先需要存储任务对象中的数据和UI对象的引用；接下来传递任务对象和状态码给实例化Handler的那个对象。在这个对象里，发送一个包含任务对象和状态的Message给Handler也运行在UI线程中，所以它可以把数据转移到UI线程。

在任务对象中存储数据

比如这里有一个Runnable，它运行在一个编码了一个Bitmap且存储这个Bitmap到父类*PhotoTask*对象里的后台线程。这个Runnable同样也存储了状态码*DECODE_STATE_COMPLETED*。

```
// A class that decodes photo files into Bitmaps
class PhotoDecodeRunnable implements Runnable {
    ...
    PhotoDecodeRunnable(PhotoTask downloadTask) {
        mPhotoTask = downloadTask;
    }
    ...
    // Gets the downloaded byte array
    byte[] imageBuffer = mPhotoTask.getByteBuffer();
    ...
    // Runs the code for this task
    public void run() {
        ...
        // Tries to decode the image buffer
        returnBitmap = BitmapFactory.decodeByteArray(
                imageBuffer,
                0,
                imageBuffer.length,
                bitmapOptions
        );
        ...
        // Sets the ImageView Bitmap
        mPhotoTask.setImage(returnBitmap);
        // Reports a status of "completed"
        mPhotoTask.handleDecodeState(DECODE_STATE_COMPLETED);
        ...
    }
    ...
}
...
```

*PhotoTask*类还包含一个用于显示Bitmap的ImageView的引用。虽然Bitmap和ImageViewImageView的引用在同一个对象中，但你不能把这个Bitmap分配给ImageView去显示，因为它们并没有运行在UI线程中。

这时，下一步应该发送这个状态给 `PhotoTask` 对象。

发送状态取决于对象层次

*PhotoTask*是下一个层次更高的对象，它包含将要展示数据的编码数据和View对象的引用。它会收到一个来自*PhotoDecodeRunnable*的状态码，并把这个状态码单独传递到一个包含线程池和Handler实例的对象：

```
public class PhotoTask {
    ...
    // Gets a handle to the object that creates the thread pools
    sPhotoManager = PhotoManager.getInstance();
    ...
    public void handleDecodeState(int state) {
        int outState;
        // Converts the decode state to the overall state.
        switch(state) {
            case PhotoDecodeRunnable.DECODE_STATE_COMPLETED:
                outState = PhotoManager.TASK_COMPLETE;
                break;
            ...
        }
        ...
        // Calls the generalized state method
        handleState(outState);
    }
    ...
    // Passes the state to PhotoManager
    void handleState(int state) {
        /*
         * Passes a handle to this task and the
         * current state to the class that created
         * the thread pools
         */
        sPhotoManager.handleState(this, state);
    }
    ...
}
```

转移数据到UI

从*PhotoTask*对象那里，*PhotoManager*对象收到了一个状态码和一个*PhotoTask*对象的引用。因为状态码是*TASK_COMPLETE*，所以创建一个Message应该包含状态和任务对象，然后把它发送给Handler：

```
public class PhotoManager {
    ...
    // Handle status messages from tasks
    public void handleState(PhotoTask photoTask, int state) {
        switch (state) {
            ...
            // The task finished downloading and decoding the image
            case TASK_COMPLETE:
                /*
                 * Creates a message for the Handler
                 * with the state and the task object
                 */
                Message completeMessage =
                        mHandler.obtainMessage(state, photoTask);
                completeMessage.sendToTarget();
                break;
            ...
        }
        ...
    }
```

最终，Handler.handleMessage()会检查每个传入进来的Message，如果状态码是*TASK_COMPLETE*，这时任务就完成了，而传入的Message里的*PhotoTask*对象里同时包含一个Bitmap和一个ImageView。因为Handler.handleMessage()运行在UI线程里，所以它能安全地转移Bitmap数据给ImageView：

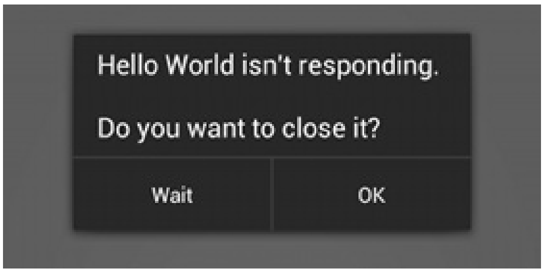
```
private PhotoManager() {
        ...
            mHandler = new Handler(Looper.getMainLooper()) {
                @Override
                public void handleMessage(Message inputMessage) {
                    // Gets the task from the incoming Message object.
                    PhotoTask photoTask = (PhotoTask) inputMessage.obj;
                    // Gets the ImageView for this task
                    PhotoView localView = photoTask.getPhotoView();
                    ...
                    switch (inputMessage.what) {
                        ...
                        // The decoding is done
                        case TASK_COMPLETE:
                            /*
                             * Moves the Bitmap from the task
                             * to the View
                             */
                            localView.setImageBitmap(photoTask.getImage());
                            break;
                        ...
                        default:
                            /*
                             * Pass along other messages from the UI
                             */
                            super.handleMessage(inputMessage);
                    }
                    ...
                }
                ...
            }
            ...
    }
...
}
```

避免出现程序无响应ANR(Keeping Your App Responsive)

> 编写:kesenhoo - 原文:http://developer.android.com/training/articles/perf-anr.html

可能你写的代码在性能测试上表现良好，但是你的应用仍然有时候会反应迟缓(sluggish)，停顿(hang)或者长时间卡死(frezze)，或者是应用处理输入的数据花费时间过长。对于你的应用来说最槽糕的事情是出现"程序无响应(Application Not Responding)" (ANR)的警示框。

在Android中，系统通过显示ANR警示框来保护程序的长时间无响应。对话框如下：



此时，你的应用已经经历过一段时间的无法响应了，因此系统提供用户可以退出应用的选择。为你的程序提供良好的响应性是至关重要的，这样才能够避免系统为用户显示ANR的警示框。

这节课描述了Android系统是如何判断一个应用不可响应的。这节课还会提供程序编写的指导原则，确保你的程序保持响应性。

是什么导致了ANR?(What Triggers ANR?)

通常来说，系统会在程序无法响应用户的输入事件时显示ANR。例如，如果一个程序在UI线程执行I/O操作(通常是网络请求或者是文件读写)，这样系统就无法处理用户的输入事件。或者是应用在UI线程花费了太多的时间用来建立一个复杂的在内存中的数据结构，又或者是在一个游戏程序的UI线程中执行了一个复杂耗时的计算移动的操作。确保那些计算操作高效是很重要的，不过即使是最高效的代码也是需要花时间执行的。

对于你的应用中任何可能长时间执行的操作，你都不应该执行在**UI**线程。你可以创建一个工作线程，把那些操作都执行在工作线程中。这确保了UI线程(这个线程会负责处理UI事件) 能够顺利执行，也预防了系统因代码僵死而崩溃。因为UI线程是和类级别相关联的，你可以把相应性作为一个类级别(class-level)的问题(相比来说，代码性能则属于方法级别(method-level)的问题)

在Android中，程序的响应性是由Activity Manager与Window Manager系统服务来负责监控的。当系统监测到下面的条件之一时会显示ANR的对话框:

- 对输入事件(例如硬件点击或者屏幕触摸事件)，5秒内都无响应。
- BroadReceiver不能够在10秒内结束接收到任务。

如何避免ANRs(How to Avoid ANRs)

Android程序通常是执行在默认的UI线程(也就是main线程)中的。这意味着在UI线程中执行的任何长时间的操作都可能触发ANR，因为程序没有给自己处理输入事件或者broadcast事件的机会。

因此，任何执行在UI线程的方法都应该尽可能的简短快速。特别是，在activity的生命周期的关键方法 `onCreate()` 与 `onResume()` 方法中应该尽可能的做比较少的事情。类似网络或者DB操作等可能长时间执行的操作，或者是类似调整bitmap大小等需要长时间计算的操作，都应该执行在工作线程中。(在DB操作中，可以通过异步的网络请求)。

为了执行一个长时间的耗时操作而创建一个工作线程最方便高效的方式是使用 `AsyncTask` 。只需要继承AsyncTask并实现 `doInBackground()` 方法来执行任务即可。为了把任务执行的进度呈现给用户，你可以执行 `publishProgress()` 方法，这个方法会触发 `onProgressUpdate()` 的回调方法。在 `onProgressUpdate()` 的回调方法中(它执行在UI线程)，你可以执行通知用户进度的操作，例如：

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

为了能够执行这个工作线程，只需要创建一个实例并执行 `execute()` :

```
new DownloadFilesTask().execute(url1, url2, url3);
```

相比起AsycnTask来说，创建自己的线程或者HandlerThread稍微复杂一点。如果你想这样做，你应该通过 `Process.setThreadPriority()` 并传递 `THREAD_PRIORITY_BACKGROUND` 来设置线程的优先级为**"background"**。如果你不通过这个方式来给线程设置一个低的优先级，那么这个线程仍然会使得你的应用显得卡顿，因为这个线程默认与UI线程有着同样的优先级。

如果你实现了Thread或者HandlerThread，请确保你的UI线程不会因为等待工作线程的某个任务而去执行Thread.wait()或者Thread.sleep()。UI线程不应该去等待工作线程完成某个任务，你的UI线程应该提供一个Handler给其他工作线程，这样工作线程能够通过这个Handler在任务结束的时候通知UI线程。使用这样的方式来设计你的应用程序可以使得你的程序UI线程保持响应性，以此来避免ANR。

BroadcastReceiver有特定执行时间的限制说明了broadcast receivers应该做的是：简短快速的任务，避免执行费时的操作，例如保存数据或者注册一个Notification。正如在UI线程中执行的方法一样，程序应该避免在broadcast receiver中执行费时的长任务。但不是采用通过工作线程来执行复杂的任务的方式，你的程序应该启动一个IntentService来响应intent broadcast的长时间任务。

> **Tip:** 你可以使用StrictMode来帮助寻找因为不小心加入到UI线程的潜在的长时间执行的操作，例如网络或者DB相关的任务。

增加响应性(Reinforce Responsiveness)

通常来说，100ms - 200ms是用户能够察觉到卡顿的上限。这样的话，下面有一些避免ANR的技巧：

- 如果你的程序需要响应正在后台加载的任务，在你的UI中可以显示ProgressBar来显示进度。
- 对游戏程序，在工作线程执行计算的任务。
- 如果你的程序在启动阶段有一个耗时的初始化操作，可以考虑显示一个闪屏，要么尽快的显示主界面，然后马上显示一个加载的对话框，异步加载数据。无论哪种情况，你都应该显示一个进度信息，以免用户感觉程序有卡顿的情况。
- 使用性能测试工具，例如Systrace与Traceview来判断程序中影响响应性的瓶颈。

JNI Tips

> 编写:pedant - 原文:http://developer.android.com/training/articles/perf-jni.html

JNI全称Java Native Interface。它为托管代码（使用Java编程语言编写）与本地代码（使用C/C++编写）提供了一种交互方式。它是与厂商无关的（vendor-neutral）,支持从动态共享库中加载代码，虽然这样会稍显麻烦，但有时这是相当有效的。

如果你对JNI还不是太熟悉，可以先通读Java Native Interface Specification这篇文章来对JNI如何工作以及哪些特性可用有个大致的印象。这种接口的一些方面不能立即一读就显而易见，所以你会发现接下来的几个章节很有用处。

JavaVM 及 JNIEnv

JNI定义了两种关键数据结构，“JavaVM”和“JNIEnv”。它们本质上都是指向函数表指针的指针（在C++版本中，它们被定义为类，该类包含一个指向函数表的指针，以及一系列可以通过这个函数表间接地访问对应的JNI函数的成员函数）。JavaVM提供“调用接口（invocation interface）”函数，允许你创建和销毁一个JavaVM。理论上你可以在一个进程中拥有多个JavaVM对象，但安卓只允许一个。

JNIEnv提供了大部分JNI功能。你定义的所有本地函数都会接收JNIEnv作为第一个参数。

JNIEnv是用作线程局部存储。因此，你不能在线程间共享一个**JNIEnv**变量。如果在一段代码中没有其它办法获得它的JNIEnv，你可以共享JavaVM对象，使用GetEnv来取得该线程下的JNIEnv（如果该线程有一个JavaVM的话；见下面的AttachCurrentThread）。

JNIEnv和JavaVM的在C声明是不同于在C++的声明。头文件“jni.h”根据它是以C还是以C++模式包含来提供不同的类型定义（typedefs）。因此，不建议把JNIEnv参数放到可能被两种语言引入的头文件中（换一句话说：如果你的头文件需要#ifdef __cplusplus，你可能不得不在任何涉及到JNIEnv的内容处都要做些额外的工作）。

线程

所有的线程都是Linux线程，由内核统一调度。它们通常从托管代码中启动（使用Thread.start），但它们也能够在其他任何地方创建，然后连接（attach）到JavaVM。例如，一个用pthread_create启动的线程能够使用JNI AttachCurrentThread 或 AttachCurrentThreadAsDaemon函数连接到JavaVM。在一个线程成功连接（attach）之前，它没有JNIEnv，不能够调用**JNI**函数。

连接一个本地环境创建的线程会触发构造一个java.lang.Thread对象，然后其被添加到主线程群组（main ThreadGroup）,以让调试器可以探测到。对一个已经连接的线程使用AttachCurrentThread不做任何操作（no-op）。

安卓不能中止正在执行本地代码的线程。如果正在进行垃圾回收，或者调试器已发出了中止请求，安卓会在下一次调用JNI函数的时候中止线程。

连接过的（attached）线程在它们退出之前必须通过**JNI**调用**DetachCurrentThread**。如果你觉得直接这样编写不太优雅，在安卓2.0（Eclair）及以上， 你可以使用pthread_key_create来定义一个析构函数，它将会在线程退出时被调用，你可以在那儿调用DetachCurrentThread（使用生成的key与pthread_setspecific将JNIEnv存储到线程局部空间内；这样JNIEnv能够作为参数传入到析构函数当中去）。

jclass, jmethodID, jfieldID

如果你想在本地代码中访问一个对象的字段（field）,你可以像下面这样做：

- 对于类，使用FindClass获得类对象的引用
- 对于字段，使用GetFieldId获得字段ID
- 使用对应的方法（例如GetIntField）获取字段下面的值

类似地，要调用一个方法，你首先得获得一个类对象的引用，然后是方法ID（method ID）。这些ID通常是指向运行时内部数据结构。查找到它们需要些字符串比较，但一旦你实际去执行它们获得字段或者做方法调用是非常快的。

如果性能是你看重的，那么一旦查找出这些值之后在你的本地代码中缓存这些结果是非常有用的。因为每个进程当中的JavaVM是存在限制的，存储这些数据到本地静态数据结构中是非常合理的。

类引用（class reference），字段ID（field ID）以及方法ID（method ID）在类被卸载前都是有效的。如果与一个类加载器（ClassLoader）相关的所有类都能够被垃圾回收，但是这种情况在安卓上是罕见甚至不可能出现，只有这时类才被卸载。注意虽然jclass是一个类引用，但是必须要调用**NewGlobalRef**保护起来（见下个章节）。

当一个类被加载时如果你想缓存些ID，而后当这个类被卸载后再次载入时能够自动地更新这些缓存ID，正确做法是在对应的类中添加一段像下面的代码来初始化这些ID：

```
/*
 * 我们在一个类初始化时调用本地方法来缓存一些字段的偏移信息
 * 这个本地方法查找并缓存你感兴趣的class/field/method ID
 * 失败时抛出异常
 */
private static native void nativeInit();

static {
    nativeInit();
}
```

在你的C/C++代码中创建一个nativeClassInit方法以完成ID查找的工作。当这个类被初始化时这段代码将会执行一次。当这个类被卸载后而后再次载入时，这段代码将会再次执行。

局部和全局引用

每个传入本地方法的参数，以及大部分JNI函数返回的每个对象都是“局部引用”。这意味着它只在当前线程的当前方法执行期间有效。即使这个对象本身在本地方法返回之后仍然存在，这个引用也是无效的。

这同样适用于所有jobject的子类，包括jclass，jstring，以及jarray（当JNI扩展检查是打开的时候，运行时会警告你对大部分对象引用的误用）。

如果你想持有一个引用更长的时间，你就必须使用一个全局（“global”）引用了。NewGlobalRef函数以一个局部引用作为参数并且返回一个全局引用。全局引用能够保证在你调用DeleteGlobalRef前都是有效的。

这种模式通常被用在缓存一个从FindClass返回的jclass对象的时候，例如：

```
jclass localClass = env->FindClass("MyClass");
jclass globalClass = reinterpret_cast<jclass>(env->NewGlobalRef(localClass));
```

所有的JNI方法都接收局部引用和全局引用作为参数。相同对象的引用却可能具有不同的值。例如，用相同对象连续地调用NewGlobalRef得到返回值可能是不同的。为了检查两个引用是否指向的是同一个对象，你必须使用**IsSameObject**函数。绝不要在本地代码中用==符号来比较两个引用。

得出的结论就是你绝不要在本地代码中假定对象的引用是常量或者是唯一的。代表一个对象的32位值从方法的一次调用到下一次调用可能有不同的值。在连续的调用过程中两个不同的对象却可能拥有相同的32位值。不要使用jobject的值作为key.

开发者需要“不过度分配”局部引用。在实际操作中这意味着如果你正在创建大量的局部引用，或许是通过对象数组，你应该使用DeleteLocalRef手动地释放它们，而不是寄希望JNI来为你做这些。实现上只预留了16个局部引用的空间，所以如果你需要更多，要么你删掉以前的，要么使用EnsureLocalCapacity/PushLocalFrame来预留更多。

注意jfieldID和jmethodID是映射类型（opaque types），不是对象引用，不应该被传入到NewGlobalRef。原始数据指针，像GetStringUTFChars和GetByteArrayElements的返回值，也都不是对象（它们能够在线程间传递，并且在调用对应的Release函数之前都是有效的）。

还有一种不常见的情况值得一提，如果你使用AttachCurrentThread连接（attach）了本地进程，正在运行的代码在线程分离（detach）之前决不会自动释放局部引用。你创建的任何局部引用必须手动删除。通常，任何在循环中创建局部引用的本地代码可能都需要做一些手动删除。

UTF-8、UTF-16 字符串

Java编程语言使用UTF-16格式。为了便利，JNI也提供了支持变形UTF-8（Modified UTF-8）的方法。这种变形编码对于C代码是非常有用的，因为它将\u0000编码成0xc0 0x80，而不是0x00。最惬意的事情是你能在具有C风格的以\0结束的字符串上计数，同时兼容标准的libc字符串函数。不好的一面是你不能传入随意的UTF-8数据到JNI函数而还指望它正常工作。

如果可能的话，直接操作UTF-16字符串通常更快些。安卓当前在调用GetStringChars时不需要拷贝，而GetStringUTFChars需要一次分配并且转换为UTF-8格式。注意**UTF-16**字符串不是以零终止字符串，\u0000是被允许的，所以你需要像对jchar指针一样地处理字符串的长度。

不要忘记**Release**你**Get**的字符串。这些字符串函数返回jchar或者*jbyte*，都是指向基本数据类型的C格式的指针而不是局部引用。它们在Release调用之前都保证有效，这意味着当本地方法返回时它们并不主动释放。

传入**NewStringUTF**函数的数据必须是变形**UTF-8**格式。一种常见的错误情况是，从文件或者网络流中读取出的字符数据，没有过滤直接使用NewStringUTF处理。除非你确定数据是7位的ASCII格式，否则你需要剔除超出7位ASCII编码范围（high-ASCII）的字符或者将它们转换为对应的变形UTF-8格式。如果你没那样做，UTF-16的转换结果可能不会是你想要的结果。JNI扩展检查将会扫描字符串，然后警告你那些无效的数据，但是它们将不会发现所有潜在的风险。

原生类型数组

JNI提供了一系列函数来访问数组对象中的内容。对象数组的访问只能一次一条，但如果原生类型数组以C方式声明，则能够直接进行读写。

为了让接口更有效率而不受VM实现的制约，GetArrayElements系列调用允许运行时返回一个指向实际元素的指针，或者是分配些内存然后拷贝一份。不论哪种方式，返回的原始指针在相应的Release调用之前都保证有效（这意味着，如果数据没被拷贝，实际的数组对象将会受到牵制，不能重新成为整理堆空间的一部分）。你必须释放（**Release**）每个你通过**Get**得到的数组。同时，如果Get调用失败，你必须确保你的代码在之后不会去尝试调用Release来释放一个空指针（NULL pointer）。

你可以用一个非空指针作为isCopy参数的值来决定数据是否会被拷贝。这相当有用。

Release类的函数接收一个mode参数，这个参数的值可选的有下面三种。而运行时具体执行的操作取决于它返回的指针是指向真实数据还是拷贝出来的那份。

- 0
  - 真实的：实际数组对象不受到牵制
  - 拷贝的：数据将会复制回去，备份空间将会被释放。
- JNI_COMMIT
  - 真实的：不做任何操作
  - 拷贝的：数据将会复制回去，备份空间将不会被释放。
- JNI_ABORT
  - 真实的：实际数组对象不受到牵制.之前的写入不会被取消。
  - 拷贝的：备份空间将会被释放；里面所有的变更都会丢失。

检查isCopy标识的一个原因是对一个数组做出变更后确认你是否需要传入JNI_COMMIT来调用Release函数。如果你交替地执行变更和读取数组内容的代码，你也许可以跳过无操作（no-op）的JNI_COMMIT。检查这个标识的另一个可能的原因是使用JNI_ABORT可以更高效。例如，你也许想得到一个数组，适当地修改它，传入部分到其他函数中，然后丢掉这些修改。如果你知道JNI是为你做了一份新的拷贝，就没有必要再创建另一份“可编辑的（editable）”的拷贝了。如果JNI传给你的是原始数组，这时你就需要创建一份你自己的拷贝了。

另一个常见的错误（在示例代码中出现过）是认为当isCopy是false时你就可以不调用Release。实际上是没有这种情况的。如果没有分配备份空间，那么初始的内存空间会受到牵制，位置不能被垃圾回收器移动。

另外注意JNI_COMMIT标识没有释放数组，你最终需要使用一个不同的标识再次调用Release。

区间数组

当你想做的只是拷出或者拷进数据时，可以选择调用像GetArrayElements和GetStringChars这类非常有用的函数。想想下面：

```
jbyte* data = env->GetByteArrayElements(array, NULL);
if (data != NULL) {
    memcpy(buffer, data, len);
    env->ReleaseByteArrayElements(array, data, JNI_ABORT);
}
```

这里获取到了数组，从当中拷贝出开头的len个字节元素，然后释放这个数组。根据代码的实现，Get函数将会牵制或者拷贝数组的内容。上面的代码拷贝了数据（为了可能的第二次），然后调用Release；这当中JNI_ABORT确保不存在第三份拷贝了。

另一种更简单的实现方式：

```
env->GetByteArrayRegion(array, 0, len, buffer);
```

这种方式有几个优点：

- 只需要调用一个JNI函数而是不是两个，减少了开销。
- 不需要指针或者额外的拷贝数据。
- 减少了开发人员犯错的风险-在某些失败之后忘记调用Release不存在风险。

类似地，你能使用SetArrayRegion函数拷贝数据到数组，使用GetStringRegion或者GetStringUTFRegion从String中拷贝字符。

异常

当异常发生时你一定不能调用大部分的**JNI**函数。你的代码收到异常（通过函数的返回值，ExceptionCheck，或者ExceptionOccurred），然后返回，或者清除异常，处理掉。

当异常发生时你被允许调用的JNI函数有：

- DeleteGlobalRef
- DeleteLocalRef
- DeleteWeakGlobalRef
- ExceptionCheck
- ExceptionClear
- ExceptionDescribe
- ExceptionOccurred
- MonitorExit

- PopLocalFrame
- PushLocalFrame
- ReleaseArrayElements
- ReleasePrimitiveArrayCritical
- ReleaseStringChars
- ReleaseStringCritical
- ReleaseStringUTFChars

许多JNI调用能够抛出异常，但通常提供一种简单的方式来检查失败。例如，如果NewString返回一个非空值，你不需要检查异常。然而，如果你调用一个方法（使用一个像CalllObjectMethod的函数），你必须一直检查异常，因为当一个异常抛出时它的返回值将不会是有效的。

注意中断代码抛出的异常不会展开本地调用堆栈信息，Android也还不支持C++异常。JNI Throw和ThrowNew指令仅仅是在当前线程中放入一个异常指针。从本地代码返回到托管代码时，异常将会被注意到，得到适当的处理。

本地代码能够通过调用ExceptionCheck或者ExceptionOccurred捕获到异常，然后使用ExceptionClear清除掉。通常，抛弃异常而不处理会导致些问题。

没有内建的函数来处理Throwable对象自身，因此如果你想得到异常字符串，你需要找出Throwable Class，然后查找到getMessage "()Ljava/lang/String;"的方法ID，调用它，如果结果非空，使用GetStringUTFChars，得到的结果你可以传到printf(3) 或者其它相同功能的函数输出。

扩展检查

JNI的错误检查很少。错误发生时通常会导致崩溃。Android也提供了一种模式，叫做CheckJNI，这当中JavaVM和JNIEnv函数表指针被换成了函数表，它在调用标准实现之前执行了一系列扩展检查的。

额外的检查包括：

- 数组：试图分配一个长度为负的数组。
- 坏指针：传入一个不完整jarray/jclass/jobject/jstring对象到JNI函数，或者调用JNI函数时使用空指针传入到一个不能为空的参数中去。
- 类名：传入了除“java/lang/String”之外的类名到JNI函数。
- 关键调用：在一个“关键的(critical)”get和它对应的release之间做出JNI调用。
- 直接的ByteBuffers：传入不正确的参数到NewDirectByteBuffer。
- 异常：当一个异常发生时调用了JNI函数。
- JNIEnvs：在错误的线程中使用一个*JNIEnv*。
- jfieldIDs：使用一个空jfieldID，或者使用jfieldID设置了一个错误类型的值到字段（比如

说，试图将一个StringBuilder赋给String类型的域），或者使用一个静态字段下的jfieldID设置到一个实例的字段（instance field）反之亦然，或者使用的一个类的jfieldID却来自另一个类的实例。

- jmethodIDs：当调用Call*Method函数时时使用了类型错误的jmethodID：不正确的返回值，静态/非静态的不匹配，this的类型错误（对于非静态调用）或者错误的类（对于静态类调用）。
- 引用：在类型错误的引用上使用了DeleteGlobalRef/DeleteLocalRef。
- 释放模式：调用release使用一个不正确的释放模式（其它非 0，JNI_ABORT，JNI_COMMIT的值）。
- 类型安全：从你的本地代码中返回了一个不兼容的类型（比如说，从一个声明返回String的方法却返回了StringBuilder）。
- UTF-8：传入一个无效的变形UTF-8字节序列到JNI调用。

（方法和域的可访问性仍然没有检查：访问限制对于本地代码并不适用。）

有几种方法去启用CheckJNI。

如果你正在使用模拟器，CheckJNI默认是打开的。

如果你有一台root过的设备，你可以使用下面的命令序列来重启运行时（runtime），启用CheckJNI。

```
adb shell stop
adb shell setprop dalvik.vm.checkjni true
adb shell start
```

随便哪一种，当运行时（runtime）启动时你将会在你的日志输出中见到如下的字符：

```
D AndroidRuntime: CheckJNI is ON
```

如果你有一台常规的设备，你可以使用下面的命令：

```
adb shell setprop debug.checkjni 1
```

这将不会影响已经在运行的app，但是从那以后启动的任何app都将打开CheckJNI(改变属性为其它值或者只是重启都将会再次关闭CheckJNI)。这种情况下，你将会在下一次app启动时，在日志输出中看到如下字符：

```
D Late-enabling CheckJNI
```

本地库

你可以使用标准的System.loadLibrary方法来从共享库中加载本地代码。在你的本地代码中较好的做法是：

- 在一个静态类初始化时调用System.loadLibrary（见之前的一个例子中，当中就使用了nativeClassInit）。参数是“未加修饰（undecorated）”的库名称，因此要加载“libfubar.so”，你需要传入“fubar”。
- 提供一个本地函数：**jint JNI_OnLoad(JavaVM *vm, void* reserved)**
- 在JNI_OnLoad中，注册所有你的本地方法。你应该声明方法为“静态的（static）”因此名称不会占据设备上符号表的空间。

JNI_OnLoad函数在C++中的写法如下：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env;
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK) {
        return -1;
    }

    // 使用env->FindClass得到jclass
    // 使用env->RegisterNatives注册本地方法

    return JNI_VERSION_1_6;
}
```

你也可以使用共享库的全路径来调用System.load。对于Android app，你也许会发现从context对象中得到应用私有数据存储的全路径是非常有用的。

上面是推荐的方式，但不是仅有的实现方式。显式注册不是必须的，提供一个JNI_OnLoad函数也不是必须的。你可以使用基于特殊命名的“发现（discovery）”模式来注册本地方法（更多细节见：JNI spec），虽然这并不可取。因为如果一个方法的签名错误，在这个方法实际第一次被调用之前你是不会知道的。

关于JNI_OnLoad另一点注意的是：任何你在JNI_OnLoad中对FindClass的调用都发生在用作加载共享库的类加载器的上下文（context）中。一般FindClass使用与“调用栈”顶部方法相关的加载器，如果当中没有加载器（因为线程刚刚连接）则使用“系统（system）”类加载器。这就使得JNI_OnLoad成为一个查寻及缓存类引用很便利的地方。

64位机问题

Android当前设计为运行在32位的平台上。理论上它也能够构建为64位的系统，但那不是现在的目标。当与本地代码交互时，在大多数情况下这不是你需要担心的，但是如果你打算存储指针变量到对象的整型字段（integer field）这样的本地结构中，这就变得非常重要了。为了支持使用64位指针的架构，你需要使用**long**类型而不是**int**类型的字段来存储你的本地指针。

不支持的特性/向后兼容性

除了下面的例外，支持所有的JNI 1.6特性：

- DefineClass没有实现。Android不使用Java字节码或者class文件，因此传入二进制class数据将不会有效。

对Android以前老版本的向后兼容性，你需要注意：

- 本地函数的动态查找 在Android 2.0(Eclair)之前，在搜索方法名称时，字符"$"不会转换为对应的"_00024"。要使它正常工作需要使用显式注册方式或者将本地方法的声明移出内部类。
- 分离线程 在Android 2.0(Eclair)之前，使用pthread_key_create析构函数来避免"退出前线程必须分离"检查是不可行的（运行时(runtime)也使用了一个pthread key析构函数，因此这是一场看谁先被调用的竞赛）。
- 全局弱引用 在Android 2.0(Eclair)之前，全局弱引用没有被实现。如果试图使用它们，老版本将完全不兼容。你可以使用Android平台版本号常量来测试系统的支持性。 在Android 4.0 (Ice Cream Sandwich)之前，全局弱引用只能传给NewLocalRef, NewGlobalRef, 以及DeleteWeakGlobalRef（强烈建议开发者在使用全局弱引用之前都为它们创建强引用hard reference，所以这不应该在所有限制当中）。 从Android 4.0 (Ice Cream Sandwich)起，全局弱引用能够像其它任何JNI引用一样使用了。
- 局部引用 在Android 4.0 (Ice Cream Sandwich)之前，局部引用实际上是直接指针。Ice Cream Sandwich为了更好地支持垃圾回收添加了间接指针，但这并不意味着很多JNI bug在老版本上不存在。更多细节见JNI Local Reference Changes in ICS。
- 使用GetObjectRefType获得引用类型 在Android 4.0 (Ice Cream Sandwich)之前，使用直接指针（见上面）的后果就是正确地实现GetObjectRefType是不可能的。我们可以使用依次检测全局弱引用表，参数，局部表，全局表的方式来代替。第一次匹配到你的直接指针时，就表明你的引用类型是当前正在检测的类型。这意味着，例如，如果你在一个全局jclass上使用GetObjectRefType，而这个全局jclass碰巧与作为静态本地方法的隐式参数传入的jclass一样的，你得到的结果是JNILocalRefType而不是JNIGlobalRefType。

FAQ: 为什么出现了UnsatisfiedLinkError?

当使用本地代码开发时经常会见到像下面的错误：

```
java.lang.UnsatisfiedLinkError: Library foo not found
```

有时候这表示和它提示的一样---未找到库。但有些时候库确实存在但不能被dlopen(3)找开，更多的失败信息可以参见异常详细说明。

你遇到“library not found”异常的常见原因可能有这些：

- 库文件不存在或者不能被app访问到。使用adb shell ls -l 检查它的存在性和权限。
- 库文件不是用NDK构建的。这就导致设备上并不存在它所依赖的函数或者库。

另一种UnsatisfiedLinkError错误像下面这样：

```
java.lang.UnsatisfiedLinkError: myfunc
        at Foo.myfunc(Native Method)
        at Foo.main(Foo.java:10)
```

在日志中，你会发现：

```
W/dalvikvm(  880): No implementation found for native LFoo;.myfunc ()V
```

这意味着运行时尝试匹配一个方法但是没有成功，这种情况常见的原因有：

- 库文件没有得到加载。检查日志输出中关于库文件加载的信息。
- 由于名称或者签名错误，方法不能匹配成功。这通常是由于：
    - 对于方法的懒查寻，使用 extern "C"和对应的可见性（JNIEXPORT）来声明C++函数没有成功。注意Ice Cream Sandwich之前的版本，JNIEXPORT宏是不正确的，因此对新版本的GCC使用旧的jni.h头文件将不会有效。你可以使用arm-eabi-nm查看它们出现在库文件里的符号。如果它们看上去比较凌乱（像_Z15Java_Foo_myfuncP7_JNIEnvP7_jclass这样而不是Java_Foo_myfunc），或者符号类型是小写的“t”而不是一个大写的“T”,这时你就需要调整声明了。
    - 对于显式注册，在进行方法签名时可能犯了些小错误。确保你传入到注册函数的签名能够完全匹配上日志文件里提示的。记住“B”是byte，“Z”是boolean。在签名中类名组件是以“L”开头的，以“;”结束的，使用“/”来分隔包名/类名，使用“$”符来分隔内部类名称（比如说，Ljava/util/Map$Entry;）。

使用javah来自动生成JNI头文件也许能帮助你避免这些问题。

FAQ: 为什么FindClass不能找到我的类?

确保类名字符串有正确的格式。JNI类名称以包名开始，然后使用左斜杠来分隔，比如java/lang/String。如果你正在查找一个数组类，你需要以对应数目的综括号开头，使用“L”和“;”将类名两头包起来，所以一个一维字符串数组应该写成[Ljava/lang/String;。

如果类名称看上去正确，你可能运行时遇到了类加载器的问题。FindClass想在与你代码相关的类加载器中开始查找指定的类。检查调用堆栈，可能看起像：

```
Foo.myfunc(Native Method)
Foo.main(Foo.java:10)
dalvik.system.NativeStart.main(Native Method)
```

最顶层的方法是Foo.myfunc。FindClass找到与类Foo相关的ClassLoader对象然后使用它。

这通常正是你所想的。如果你创建了自己的线程那么就会遇到麻烦（也许是调用了pthread_create然后使用AttachCurrentThread进行了连接）。现在跟踪堆栈可能像下面这样：

```
dalvik.system.NativeStart.run(Native Method)
```

最顶层的方法是NativeStart.run，它不是你应用内的方法。如果你从这个线程中调用FindClass，JavaVM将会启动“系统（system）”的而不是与你应用相关的加载器，因此试图查找应用内定义的类都将会失败。

下面有几种方法可以解决这个问题：

- 在JNI_OnLoad中使用FindClass查寻一次，然后为后面的使用缓存这些类引用。任何在JNI_OnLoad当中执行的FindClass调用都使用与执行System.loadLibrary的函数相关的类加载器（这个特例，让库的初始化更加的方便了）。如果你的app代码正在加载库文件，FindClass将会使用正确的类加载器。
- 传入类实例到一个需要它的函数，你的本地方法声明必须带有一个Class参数，然后传入Foo.class。
- 在合适的地方缓存一个ClassLoader对象的引用，然后直接发起loadClass调用。这需要额外些工作。

FAQ: 使用本地代码怎样共享原始数据?

也许你会遇到这样一种情况，想从你的托管代码或者本地代码访问一大块原始数据的缓冲区。常见例子包括对bitmap或者声音文件的处理。这里有两种基本实现方式。

你可以将数据存储到byte[]。这允许你从托管代码中快速地访问。然而，在本地代码端不能保证你不去拷贝一份就直接能够访问数据。在某些实现中，GetByteArrayElements和GetPrimitiveArrayCritical将会返回指向在维护堆中的原始数据的真实指针，但是在另外一些实现中将在本地堆空间分配一块缓冲区然后拷贝数据过去。

还有一种选择是将数据存储在一块直接字节缓冲区（direct byte buffer），可以使用java.nio.ByteBuffer.allocateDirect或者NewDirectByteBuffer JNI函数创建buffer。不像常规的byte缓冲区，它的存储空间将不会分配在程序维护的堆空间上，总是可以从本地代码直接访问（使用GetDirectBufferAddress得到地址）。依赖于直接字节缓冲区访问的实现方式，从托管代码访问原始数据将会非常慢。

选择使用哪种方式取决于两个方面：

1.大部分的数据访问是在Java代码还是C/C++代码中发生？

2.如果数据最终被传到系统API，那它必须是怎样的形式（例如，如果数据最终被传到一个使用byte[]作为参数的函数，在直接的ByteBuffer中处理或许是不明智的）？

如果通过上面两种情况仍然不能明确区分的，就使用直接字节缓冲区（direct byte buffer）形式。它们的支持是直接构建到JNI中的，在未来的版本中性能可能会得到提升。

SMP(Symmetric Multi-Processor) Primer for Android

> 编写:kesenhoo - 原文:http://developer.android.com/training/articles/smp.html

从Android 3.0开始，系统针对多核CPU架构的机器做了优化支持。这份文档介绍了针对多核系统应该如何编写C，C++以及Java程序。这里只是作为Android应用开发者的入门教程，并不会深入讨论这个话题，并且我们会把讨论范围集中在ARM架构的CPU上。

如果你并没有时间学习整篇文章，你可以跳过前面的理论部分，直接查看实践部分。但是我们并不建议这样做。

0)简要介绍

SMP 的全称是“**Symmetric Multi-Processor**”。 它表示的是一种双核或者多核CPU的设计架构。在几年前，所有的Android设备都还是单核的。

大多数的Android设备已经有了多个CPU，但是通常来说，其中一个CPU负责执行程序，其他的CPU则处理设备硬件的相关事务（例如，音频）。这些CPU可能有着不同的架构，运行在上面的程序无法在内存中彼此进行沟通交互。

目前大多数售卖的Android设备都是SMP架构的，这使得软件开发者处理问题更加复杂。对于多线程的程序，如果多个线程执行在不同的内核上，这会使得程序更加容易发生**race conditions**。 更糟糕的是，基于ARM架构的SMP比起x86架构来说，更加复杂，更难进行处理。那些在x86上测试通过的程序可能会在ARM上崩溃。

下面我们会介绍为何会这样以及如何做才能够使得你的代码行为正常。

1)理论篇

这里会快速并且简要的介绍这个复杂的主题。其中一些部分并不完整，但是并没有出现错误或者误导。

查看文章末尾的进一步阅读可以了解这个主题的更多知识。

1.1)内存一致性模型(Memory consistency models)

内存一致性模型(Memory consistency models)通常也被叫做“memory models”，描述了硬件架构如何确保内存访问的一致性。例如，如果你对地址A进行了一个赋值，然后对地址B也进行了赋值，那么内存一致性模型就需要确保每一个CPU都需要知道刚才的操作赋值与操作顺序。

这个模型通常被程序员称为：顺序一致性**(sequential consistency)**, 请从文章末尾的进一步阅读查看**Adve & Gharachorloo**这篇文章。

- 所有的内存操作每次只能执行一个。
- 所有的操作，在单核CPU上，都是顺序执行的。

如果你关注一段代码在内存中的读写操作，在sequentially-consistent的CPU架构上，是按照期待的顺序执行的。It’s possible that the CPU is actually reordering instructions and delaying reads and writes, but there is no way for code running on the device to tell that the CPU is doing anything other than execute instructions in a straightforward manner. (We’re ignoring memory-mapped device driver I/O for the moment.)

To illustrate these points it’s useful to consider small snippets of code, commonly referred to as litmus tests. These are assumed to execute in program order, that is, the order in which the instructions appear here is the order in which the CPU will execute them. We don’t want to consider instruction reordering performed by compilers just yet.

Here’s a simple example, with code running on two threads:

Thread 1 Thread 2 A = 3 B = 5 reg0 = B reg1 = A

Thread 1	Thread 2
A = 3 B = 5	reg0 = B reg1 = A

In this and all future litmus examples, memory locations are represented by capital letters (A, B, C) and CPU registers start with “reg”. All memory is initially zero. Instructions are executed from top to bottom. Here, thread 1 stores the value 3 at location A, and then the value 5 at location B. Thread 2 loads the value from location B into reg0, and then loads the value from location A into reg1. (Note that we’re writing in one order and reading in another.)

Thread 1 and thread 2 are assumed to execute on different CPU cores. You should always make this assumption when thinking about multi-threaded code.

Sequential consistency guarantees that, after both threads have finished executing, the registers will be in one of the following states:

Registers	States
reg0=5, reg1=3	possible (thread 1 ran first)
reg0=0, reg1=0	possible (thread 2 ran first)
reg0=0, reg1=3	possible (concurrent execution)
reg0=5, reg1=0	never

To get into a situation where we see B=5 before we see the store to A, either the reads or the writes would have to happen out of order. On a sequentially-consistent machine, that can't happen.

Most uni-processors, including x86 and ARM, are sequentially consistent. Most SMP systems, including x86 and ARM, are not.

1.1.1)Processor consistency

1.1.2)CPU cache behavior

1.1.3)Observability

1.1.4)ARM's weak ordering

1.2)Data memory barriers

1.2.1)Store/store and load/load

1.2.2)Load/store and store/load

1.2.3)Barrier instructions

1.2.4)Address dependencies and causal consistency

1.2.5)Memory barrier summary

1.3)Atomic operations

1.3.1)Atomic essentials

1.3.2)Atomic + barrier pairing

1.3.3)Acquire and release

2)实践篇

调试内存一致性(memory consistency)的问题非常困难。如果内存栅栏(memory barrier)导致一些代码读取到陈旧的数据，你将无法通过调试器检查内存dumps文件来找出原因。By the time you can issue a debugger query, the CPU cores will have all observed the full set of accesses, and the contents of memory and the CPU registers will appear to be in an “impossible” state.

2.1)What not to do in C

2.1.1)C/C++ and “volatile”

2.1.2)Examples

2.2)在Java中不应该做的事

我们没有讨论过Java语言的一些相关特性，因此我们首先来简要的看下那些特性。

2.2.1)Java中的"synchronized"与"volatile"关键字

“synchronized”关键字提供了Java一种内置的锁机制。每一个对象都有一个相对应的“monitor”，这个监听器可以提供互斥的访问。

“synchronized”代码段的实现机制与自旋锁(spin lock)有着相同的基础结构: 他们都是从获取到CAS开始，以释放CAS结束。这意味着编译器(compilers)与代码优化器(code optimizers)可以轻松的迁移代码到“synchronized”代码段中。一个实践结果是：你不能判定synchronized代码段是执行在这段代码下面一部分的前面，还是这段代码上面一部分的后面。更进一步，如果一个方法有两个synchronized代码段并且锁住的是同一个对象，那么在这两个操作的中间代码都无法被其他的线程所检测到，编译器可能会执行“锁粗化lock coarsening”并且把这两者绑定到同一个代码块上。

另外一个相关的关键字是**“volatile”**。在Java 1.4以及之前的文档中是这样定义的：volatile声明和对应的C语言中的一样可不靠。从Java 1.5开始，提供了更有力的保障，甚至和synchronization一样具备强同步的机制。

volatile的访问效果可以用下面这个例子来说明。如果线程1给volatile字段做了赋值操作，线程2紧接着读取那个字段的值，那么线程2是被确保能够查看到之前线程1的任何写操作。更通常的情况是，任何线程对那个字段的写操作对于线程2来说都是可见的。实际上，写volatile就像是释放件监听器，读volatile就像是获取监听器。

非volatile的访问有可能因为照顾volatile的访问而需要做顺序的调整。例如编译器可能会往上移动一个非volatile加载操作，但是不会往下移动。Volatile之间的访问不会因为彼此而做出顺序的调整。虚拟机会注意处理如何的内存栅栏(memory barriers)。

当加载与保存大多数的基础数据类型，他们都是原子的atomic, 对于long以及double类型的数据则不具备原子型，除非他们被声明为volatile。即使是在单核处理器上，并发多线程更新非volatile字段值也还是不确定的。

2.2.2)Examples

下面是一个错误实现的单步计数器(monotonic counter)的示例: (Java theory and practice: Managing volatility).

```
class Counter {
    private int mValue;

    public int get() {
        return mValue;
    }
    public void incr() {
        mValue++;
    }
}
```

假设get()与incr()方法是被多线程调用的。然后我们想确保当get()方法被调用时，每一个线程都能够看到当前的数量。最引人注目的问题是mValue++实际上包含了下面三个操作。

1. reg = mValue
2. reg = reg + 1
3. mValue = reg

如果两个线程同时在执行 `incr()` 方法，其中的一个更新操作会丢失。为了确保正确的执行 `++` 的操作，我们需要把 `incr()` 方法声明为“synchronized”。这样修改之后，这段代码才能够在单核多线程的环境中正确的执行。

然而，在SMP的系统下还是会执行失败。不同的线程通过 `get()` 方法获取到得值可能是不一样的。因为我们是使用通常的加载方式来读取这个值的。我们可以通过声明 `get()` 方法为synchronized的方式来修正这个错误。通过这些修改，这样的代码才是正确的了。

不幸的是，我们有介绍过有可能发生的锁竞争(lock contention)，这有可能会伤害到程序的性能。除了声明 `get()` 方法为synchronized之外，我们可以声明 `mValue` 为**“volatile”**. (请注意 `incr()` 必须使用synchronize) 现在我们知道volatile的mValue的写操作对于后续的读操作都是可见的。 `incr()` 将会稍稍有点变慢，但是 `get()` 方法将会变得更加快速。因此读操作多于写操作时，这会是一个比较好的方案。(请参考AtomicInteger.)

下面是另外一个示例，和之前的C示例有点类似：

```
class MyGoodies {
    public int x, y;
}
class MyClass {
    static MyGoodies sGoodies;

    void initGoodies() {    // runs in thread 1
        MyGoodies goods = new MyGoodies();
        goods.x = 5;
        goods.y = 10;
        sGoodies = goods;
    }

    void useGoodies() {    // runs in thread 2
        if (sGoodies != null) {
            int i = sGoodies.x;    // could be 5 or 0
            ....
        }
    }
}
```

这段代码同样存在着问题， `sGoodies = goods` 的赋值操作有可能在 `goods` 成员变量赋值之前被察觉到。如果你使用 `volatile` 声明 `sGoodies` 变量，你可以认为load操作为 `atomic_acquire_load()` ，并且把store操作认为是 `atomic_release_store()` 。

(请注意仅仅是 `sGoodies` 的引用本身为 `volatile` ，访问它的内部字段并不是这样的。赋值语句 `z = sGoodies.x` 会执行一个volatile load MyClass.sGoodies的操作，其后会伴随一个non-volatile的load操作：: `sGoodies.x` 。如果你设置了一个本地引用 `MyGoodies localGoods = sGoodies, z = localGoods.x` ，这将不会执行任何volatile loads.)

另外一个在Java程序中更加常用的范式就是臭名昭著的**“double-checked locking”**:

```
class MyClass {
    private Helper helper = null;

    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

上面的写法是为了获得一个MyClass的单例。我们只需要创建一次这个实例，通过 `getHelper()` 这个方法。为了避免两个线程会同时创建这个实例。我们需要对创建的操作加synchronize机制。然而，我们不想要为了每次执行这段代码的时候都为“synchronized”付出额外的代价，因此我们仅仅在helper对象为空的时候加锁。

在单核系统上，这是不能正常工作的。JIT编译器会破坏这件事情。请查看4)Appendix的“‘Double Checked Locking is Broken’ Declaration”获取更多的信息, 或者是Josh Bloch’s Effective Java书中的Item 71 (“Use lazy initialization judiciously”)。

在SMP系统上执行这段代码，引入了一个额外的方式会导致失败。把上面那段代码换成C的语言实现如下：

```
if (helper == null) {
    // acquire monitor using spinlock
    while (atomic_acquire_cas(&this.lock, 0, 1) != success)
        ;
    if (helper == null) {
        newHelper = malloc(sizeof(Helper));
        newHelper->x = 5;
        newHelper->y = 10;
        helper = newHelper;
    }
    atomic_release_store(&this.lock, 0);
}
```

此时问题就更加明显了: `helper` 的store操作发生在memory barrier之前，这意味着其他的线程能够在store x/y之前观察到非空的值。

你应该尝试确保store helper执行在 `atomic_release_store()` 方法之后。通过重新排序代码进行加锁，但是这是无效的，因为往上移动的代码，编译器可以把它移动回原来的位置：在 `atomic_release_store()` 前面。(这里没有读懂，下次再回读)

有2个方法可以解决这个问题：

- 删除外层的检查。这确保了我们不会在synchronized代码段之外做任何的检查。
- 声明helper为volatile。仅仅这样一个小小的修改，在前面示例中的代码就能够在Java 1.5及其以后的版本中正常工作。

下面的示例演示了使用volatile的2各重要问题：

```
class MyClass {
    int data1, data2;
    volatile int vol1, vol2;

    void setValues() {    // runs in thread 1
        data1 = 1;
        vol1 = 2;
        data2 = 3;
    }

    void useValues1() {    // runs in thread 2
        if (vol1 == 2) {
            int l1 = data1;    // okay
            int l2 = data2;    // wrong
        }
    }
    void useValues2() {    // runs in thread 2
        int dummy = vol2;
        int l1 = data1;    // wrong
        int l2 = data2;    // wrong
    }
```

请注意 `useValues1()` ，如果thread 2还没有察觉到 `vol1` 的更新操作，那么它也无法知道 `data1` 或者 `data2` 被设置的操作。一旦它观察到了 `vol1` 的更新操作，那么它也能够知道data1的更新操作。然而，对于 `data2` 则无法做任何猜测，因为store操作是在volatile store之后发生的。

`useValues2()` 使用了第2个volatile字段：vol2，这会强制VM生成一个memory barrier。这通常不会发生。为了建立一个恰当的“happens-before”关系，2个线程都需要使用同一个volatile字段。在thread 1中你需要知道vol2是在data1/data2之后被设置的。(The fact that this doesn’t work is probably obvious from looking at the code; the caution here is against trying to cleverly “cause” a memory barrier instead of creating an ordered series of accesses.)

2.3)What to do

2.3.1)General advice

在C/C++中，使用 `pthread` 操作，例如mutexes与semaphores。他们会使用合适的memory barriers，在所有的Android平台上提供正确有效的行为。请确保正确这些技术，例如在没有获得对应的mutex的情况下赋值操作需要很谨慎。

避免直接使用atomic方法。如果locking与unlocking之间没有竞争，locking与unlocking一个pthread mutex 分别需要一个单独的atomic操作。如果你需要一个lock-free的设计，你必须在开始写代码之前了解整篇文档的要点。（或者是寻找一个已经为SMP ARM设计好的库文件）。

Be extremely circumspect with "volatile” in C/C++. It often indicates a concurrency problem waiting to happen.

In Java, the best answer is usually to use an appropriate utility class from the java.util.concurrent package. The code is well written and well tested on SMP.

Perhaps the safest thing you can do is make your class immutable. Objects from classes like String and Integer hold data that cannot be changed once the class is created, avoiding all synchronization issues. The book Effective Java, 2nd Ed. has specific instructions in “Item 15: Minimize Mutability”. Note in particular the importance of declaring fields “final" (Bloch).

If neither of these options is viable, the Java “synchronized” statement should be used to guard any field that can be accessed by more than one thread. If mutexes won’t work for your situation, you should declare shared fields “volatile”, but you must take great care to understand the interactions between threads. The volatile declaration won’t save you from common concurrent programming mistakes, but it will help you avoid the mysterious failures associated with optimizing compilers and SMP mishaps.

The Java Memory Model guarantees that assignments to final fields are visible to all threads once the constructor has finished — this is what ensures proper synchronization of fields in immutable classes. This guarantee does not hold if a partially-constructed object is allowed to become visible to other threads. It is necessary to follow safe construction practices.(Safe Construction Techniques in Java).

2.3.2)Synchronization primitive guarantees

The pthread library and VM make a couple of useful guarantees: all accesses previously performed by a thread that creates a new thread are observable by that new thread as soon as it starts, and all accesses performed by a thread that is exiting are observable when a join() on that thread returns. This means you don’t need any additional synchronization when preparing data for a new thread or examining the results of a joined thread.

Whether or not these guarantees apply to interactions with pooled threads depends on the thread pool implementation.

In C/C++, the pthread library guarantees that any accesses made by a thread before it unlocks a mutex will be observable by another thread after it locks that same mutex. It also guarantees that any accesses made before calling signal() or broadcast() on a condition variable will be observable by the woken thread.

Java language threads and monitors make similar guarantees for the comparable operations.

2.3.3)Upcoming changes to C/C++

The C and C++ language standards are evolving to include a sophisticated collection of atomic operations. A full matrix of calls for common data types is defined, with selectable memory barrier semantics (choose from relaxed, consume, acquire, release, acq_rel, seq_cst).

See the Further Reading section for pointers to the specifications.

3)Closing Notes

While this document does more than merely scratch the surface, it doesn't manage more than a shallow gouge. This is a very broad and deep topic. Some areas for further exploration:

- Learn the definitions of **happens-before**, **synchronizes-with**, and other essential concepts from the Java Memory Model. (It's hard to understand what “volatile” really means without getting into this.)
- Explore what compilers are and aren't allowed to do when reordering code. (The JSR-133 spec has some great examples of legal transformations that lead to unexpected results.)
- Find out how to write immutable classes in Java and C++. (There's more to it than just “don't change anything after construction”.)
- Internalize the recommendations in the Concurrency section of **Effective Java, 2nd Edition**. (For example, you should avoid calling methods that are meant to be overridden while inside a synchronized block.)
- Understand what sorts of barriers you can use on x86 and ARM. (And other CPUs for that matter, for example Itanium's acquire/release instruction modifiers.)
- Read through the **java.util.concurrent** and **java.util.concurrent.atomic** APIs to see what's available.
- Consider using concurrency annotations like `@ThreadSafe` and `@GuardedBy` (from net.jcip.annotations).

The Further Reading section in the appendix has links to documents and web sites that will better illuminate these topics.

4)Appendix

4.1)SMP failure example

4.2)Implementing synchronization stores

4.3)Further reading

保护安全与隐私的最佳策略

> 编写:craftsmanBai - http://z1ng.net - 原文:http://developer.android.com/training/best-security.html

下面的课程教你如何确保应用程序数据的安全。

安全要点

怎样执在执行多个任务的同时确保应用程序数据和用户数据的安全。

HTTPS和SSL的安全

如何确保应用程序在进行网络传输时是安全的。

更新你的Security Provider对抗SSL漏洞攻击

如何使用和更新Google Play services security provider来对抗SSL漏洞攻击。

企业级开发

如何为企业级应用程序实施设备管理策略。

安全要点

> 编写:craftsmanBai - http://z1ng.net - 原文:http://developer.android.com/training/articles/security-tips.html

Android内建的安全机制可以显著地减少了应用程序的安全问题。你可以在默认的系统设置和文件权限设置的环境下建立应用，避免针对一堆头疼的安全问题寻找解决方案。

一些帮助建立应用的核心安全特性如下：

- Android应用程序沙盒，将应用数据和代码的执行与其他程序隔离。
- 具有鲁棒性的常见安全功能的应用框架，例如加密，权限控制，安全IPC
- 使用ASLR，NX，ProPolice，safe_iop，OpenBSD dlmalloc，OpenBSD calloc，Linux mmap_min_addr等技术，减少了常见内存管理错误。
- 加密文件系统可以保护丢失或被盗走的设备数据。
- 用户权限控制限制访问系统关键信息和用户数据。
- 应用程序权限以单个应用为基础控制其数据。

尽管如此，熟悉Android安全特性仍然很重要。遵守这些习惯并将其作为优秀的代码风格，能够减少无意间给用户带来的安全问题。

数据存储

对于一个Android的应用程序来说，最为常见的安全问题是存放在设备上的数据能否被其他应用获取。在设备上存放数据基本方式有三种:

使用内部存储

默认情况下，你在内部存储中创建的文件只有你的应用可以访问。Android实现了这种机制，并且对于大多数应用程序都是有效的。 你应该避免在IPC文件中使用MODE_WORLD_WRITEABLE或者MODE_WORLD_READABLE模式，因为它们不为特殊程序提供限制数据访问的功能，它们也不对数据格式进行任何控制。如果你想与其他应用的进程共享数据，可以使用Content Provider，它可以给其他应用提供了可读写权限以及逐项动态获取权限。

如果想对敏感数据进行特别保护，你可以使用应用程序无法直接获取的密钥来加密本地文件。例如，密钥可以存放在KeyStore而非设备上，使用用户密码进行保护。尽管这种方式无法防止通过root权限查看用户输入的密码，但是它可以为未进行文件系统加密的丢失设备提供保护。

使用外部存储

创建于外部存储的文件，比如SD卡，是全局可读写的。 由于外部存储器可被用户移除并且能够被任何应用修改，因此不应使用外部存储保存应用的敏感信息。 当处理来自外部存储的数据时，应用程序应该执行输入验证（参看输入验证章节） 我们强烈建议应用在动态加载之前不要把可执行文件或class文件存储到外部存储中。 如果一个应用从外部存储检索可执行文件，那么在动态加载之前它们应该进行签名与加密验证。

使用Content Providers

ContentProviders提供了一种结构存储机制，它可以限制你自己的应用，也可以允许其他应用程序进行访问。 如果你不打算向其他应用提供访问你的ContentProvider功能，那么在manifest中标记他们为android:exported=false即可。 要建立一个给其他应用使用的ContentProvider，你可以为读写操作指定一个单一的permission，或者在manifest中为读写操作指定确切的权限。我们强烈建议你对要分配的权限进行限制，仅满足目前有的功能即可。记住，通常新的权限在新功能加入的时候同时增加，会比把现有权限撤销并打断已经存在的用户更合理。

如果Content Provider仅在自己的应用中共享数据，使用签名级别android:protectionLevel的权限是更可取的。 签名权限不需要用户确认，当应用使用同样的密钥获取数据时，这提供了更好的用户体验，也更好地控制了Content Provider数据的访问。 Content Providers也可以通过声明android:grantUriPermissions并在触发组件的Intent对象中使用FLAG_GRANT_READ_URI_PERMISSION和FLAG_GRANT_WRITE_URI_PERMISSION标志提供更细致的访问。 这些许可的作用域可以通过grant-uri-permission进一步限制。 当访问一个ContentProvider时，使用参数化的查询方法，比如query()，update()和delete()来避免来自不信任源潜在的SQL注入。 注意，如果selection语句是在提交给方法之前先连接用户数据的，使用参数化的方法或许不够。 不要对“写”权限有一个错误的观念。 考虑“写”权限允许sql语句，它可以通过使用创造性的WHERE子句并且解析结果让部分数据的确认变为可能。 例如：入侵者可能在通话记录中通过修改一条记录来检测某个特定存在的电话号码，只要那个电话号码已经存在。 如果content provider数据有可预见的结构，提供“写”权限也许等同于同时提供了“读写”权限。

使用权限

因为安卓沙盒将应用程序隔离，程序必须显式地共享资源和数据。它们通过声明他们需要的权限来获取额外的功能，而基本的沙盒不提供这些功能，比如相机访问设备。

请求权限

我们建议最小化应用请求的权限数量，不具有访问敏感资料的权限可以减少无意中滥用这些权限的风险，可以增加用户接受度，并且减少应用被攻击者攻击利用的可能性。

如果你的应用可以设计成不需要任何权限，那最好不过。例如：与其请求访问设备信息来建立一个标识，不如建立一个GUID（这个例子在下文"处理用户数据"中有说明）。

除了请求权限之外，你的应用可以使用permissions来保护可能会暴露给其他应用的安全敏感的IPC：比如ContentProvider。通常来说，我们建议使用访问控制而不是用户权限确认许可，因为权限会使用户感到困惑。例如，考虑在权限设置上为应用间的IPC通信使用单一开发者提供的签名保护级别。

不要泄漏受许可保护的数据。只有当应用通过IPC暴露数据才会发生这种情况，因为它具有特殊权限，却不要求任何客户端的IPC接口有那样的权限。更多关于这方面的潜在影响以及这种问题发生的频率在USENIX: http://www.cs.be rkeley.edu/~afelt/felt_usenixsec2011.pdf研究论文中都有说明。

创建权限

通常，你应该力求建立拥有尽量少权限的应用，直至满足你的安全需要。建立一个新的权限对于大多数应用相对少见，因为系统定义的许可覆盖很多情况。在适当的地方使用已经存在的许可执行访问检查。

如果必须建立一个新的权限，考虑能否使用signature protection level来完成你的任务。签名许可对用户是透明的并且只允许相同开发者签名的应用访问，与应用执行权限检查一样。如果你建立一个dagerous protction level，那么用户需要决定是否安装这个应用。这会使其他开发者困惑，也使用户困惑。

如果你要建立一个危险的许可，则会有多种复杂情况需考虑：

- 对于用户将要做出的安全决定，许可需要用字符串对其进行简短的表述。
- 许可字符串必须保证语言的国际化。
- 用户可能对一个许可感到困惑或者知晓风险而选择不安装应用
- 当许可的创造者未安装的时候，应用可能要求许可。

上面每一个因素都为应用开发者带来了重要的非技术挑战，同时也使用户感到困惑，这也是我们不建议使用危险许可的原因。

使用网络

网络交易具有很高的安全风险，因为它涉及到传送私人的数据。人们对移动设备的隐私关注日益加深，特别是当设备进行网络交易时，因此应用采取最佳方式保护用户数据安全极为重要。

使用IP网络

Android下的网络与Linux环境下的差别并不大。主要考虑的是确保对敏感数据采用了适当的协议，比如使用HTTPS进行网络传输。我们在任何支持HTTPS的服务器上更愿意使用HTTPS而不是HTTP，因为移动设备可能会频繁连接不安全的网络，比如公共WiFi热点。

授权且加密的套接层级别的通信可通过使用SSLSocket类轻松实现。考虑到Android设备使用WiFi连接不安全网络的频率，对于所有应用来说，使用安全网络是极力鼓励支持的。

我们发现部分应用使用localhost端口处理敏感的IPC。我们不鼓励这种方法，是因为这些接口可被设备上的其他应用访问。相反，你应该在可认证的地方使用Android IPC机制，例如Service（比使用回环还糟的是绑定INADDR_ANY，因为你的应用可能收到来自任何地方来的请求，我们也已经见识过了）。

一个有必要重复的常见议题是，确保不信任从HTTP或者其他不安全协议下载的数据。这包括在WebView中的输入验证和对于http的任何响应。

使用电话网络

SMS协议是Android开发者使用最频繁的电话协议，主要为用户与用户之间的通信设计，但对于想要传送数据的应用来说并不合适。由于SMS的限制性，我们强烈建议使用Google Cloud Messaging（GCM）和IP网络从web服务器发送数据消息给用户设备应用。

很多开发者没有意识到SMS在网络上或者设备上是不加密的，也没有牢固验证。特别是任何SMS接收者应该预料到恶意用户也许已经给你的应用发送了SMS：不要指望未验证的SMS数据执行敏感操作。你也应该注意到SMS在网络上也许会遭到冒名顶替并且/或者拦截，对于Android设备本身，SMS消息是通过广播intent传递的，所以他们也许会被其他拥有READ_SMS许可的应用截获。

输入验证

无论应用运行在什么平台上，功能不完善的输入验证是最常见的影响应用安全问题之一。Android有平台级别的对策，用于减少应用的公开输入验证问题，你应该在可能的地方使用这些功能。同样需要注意的是，选择类型安全的语言能减少输入验证问题。

如果你使用native代码，那么任何从文件读取的，通过网络接收的，或者通过IPC接收的数据都有可能引发安全问题。最常见的问题是buffer overflows，use after free，和off-by-one。Android提供安全机制比如ASLR和DEP以减少这些漏洞的可利用性，但是没有解决基本的问题。小心处理指针和管理缓存可以预防这些问题。

动态、基于字符串的语言，比如JavaScript和SQL，都常受到由转义字符和脚本注入带来的输入验证问题。

如果你使用提交到SQL Database或者Content Provider的数据，SQL注入也许是个问题。最好的防御是使用参数化的查询，就像ContentProviders中讨论的那样。限制权限为只读或者只写可以减少SQL注入的潜在危害。

如果你不能使用上面提到的安全功能，我们强烈建议使用结构严谨的数据格式并且验证符合期望的格式。黑名单策略与替换危险字符是有效的，但这些技术在实践中是易错的并且当错误可能发生的时候应该尽量避免。

处理用户数据

通常来说，处理用户数据安全最好的方法是最小化获取敏感数据用户个人数据的API使用。如果你对数据进行访问并且可以避免存储或传输，那就不要存储和传输数据。最后，思考是否有一种应用逻辑可能被实现为使用hash或者不可逆形式的数据。例如，你的应用也许使用一个email地址的hash作为主键，避免传输或存储email地址，这减少无意间泄漏数据的机会，并且也能减少攻击者尝试利用你的应用的机会。

如果你的应用访问私人数据，比如密码或者用户名，记住司法也许要求你提供一个使用和存储这些数据的隐私策略的解释。所以遵守最小化访问用户数据最佳的安全实践也许只是简单的服从。

你也应该考虑到应用是否会疏忽暴露个人信息给其他方，比如广告第三方组件或者你应用使用的第三方服务。如果你不知道为什么一个组件或者服务请求个人信息，那么就不要提供给它。通常来说，通过减少应用访问个人信息，会减少这个区域潜在的问题。

如果必须访问敏感数据，评估这个信息是否必须要传到服务器，或者是否可以被客户端操作。考虑客户端上使用敏感数据运行的任何代码，避免传输用户数据 确保不会无意间通过过渡自由的IPC、world writable文件、或网络socket暴露用户数据给其他设备上的应用。这里有一个泄漏权限保护数据的特别例子，在Requesting Permissions章节中讨论。

如果需要GUID，建立一个大的、唯一的数字并保存它。不要使用电话标识，比如与个人信息相关的电话号码或者IMEI。这个话题在Android Developer Blog中有更详细的讨论。

应用开发者应谨慎的把log写到机器上。在Android中，log是共享资源，一个带有READ_LOGS许可的应用可以访问。即使电话log数据是临时的并且在重启之后会擦除，不恰当地记录用户信息会无意间泄漏用户数据给其他应用。

使用WebView

因为WebView能包含HTML和JavaScript浏览网络内容，不恰当的使用会引入常见的web安全问题，比如跨站脚本攻击（JavaScript注入）。Android采取一些机制通过限制WebView的能力到应用请求功能最小化来减少这些潜在的问题。

如果你的应用没有在WebView内直接使用JavaScript，不要调用setJavaScriptEnabled()。某些样本代码使用这种方法，可能会导致在产品应用中改变用途：所以如果不需要的话移除它。默认情况下WebView不执行JavaScript，所以跨站脚本攻击不会产生。

使用addJavaScriptInterface()要特别的小心，因为它允许JavaScript执行通常保留给Android应用的操作。只把addJavaScriptInterface()暴露给可靠的输入源。如果不受信任的输入是被允许的，不受信任的JavaScript也许会执行Android方法。总得来说，我们建议只把addJavaScriptInterface()暴露给你应用内包含的JavaScript。

如果你的应用通过WebView访问敏感数据，你也许想要使用clearCache()方法来删除任何存储到本地的文件。服务端的header，比如no-cache，能用于指示应用不应该缓存特定的内容。

处理证书

通常来说，我们建议请求用户证书频率最小化--使得钓鱼攻击更明显，并且降低其成功的可能。取而代之使用授权令牌然后刷新它。

可能的情况下，用户名和密码不应该存储到设备上，而使用用户提供的用户名和密码执行初始认证，然后使用一个短暂的、特定服务的授权令牌。可以被多个应用访问的service应该使用AccountManager访问。 如果可能的话，使用AccountManager类来执行基于云的服务并且不把密码存储到设备上。

使用AccountManager获取Account之后，进入任何证书前检查CREATOR，这样你就不会因为疏忽而把证书传递给错误的应用。

如果证书只是用于你创建的应用，那么你能使用checkSignature()验证访问AccountManager的应用。或者，如果一个应用要使用证书，你可以使用KeyStore来储存。

使用加密

除了采取数据隔离，支持完整的文件系统加密，提供安全信道之外。Android提供大量加密算法来保护数据。

通常来说，尝试使用最高级别的已存在framework的实现来支持，如果你需要安全的从一个已知的位置取回一个文件，一个简单的HTTPS URI也许就足够了，并且这部分不要求任何加密知识。如果你需要一个安全信道，考虑使用HttpsURLConnection或者SSLSocket要比使用你自己的协议好。

如果你发现的确需要实现一个自定义的协议，我们强烈建议你不要自己实现加密算法。使用已经存在的加密算法，比如Cipher类中提供的AES或者RSA。

使用一个安全的随机数生成器（SecureRandom）来初始化加密密钥（KeyGenerator）。使用一个不安全随机数生成器生成的密钥严重削弱算法的优点，而且可能遭到离线攻击。

如果你需要存储一个密钥来重复使用，使用类似于KeyStore的机制，来提供长期储存和检索加密密钥的功能。

使用进程间通信

一些Android应用试图使用传统的Linux技术实现IPC，比如网络socket和共享文件。我们强烈鼓励使用Android系统IPC功能，比如Intent，Binder，Messenger和BroadcastReceiver。Android IPC机制允许你为每一个IPC机制验证连接到你的IPC和设置安全策略的应用的身份。

很多安全元素通过IPC机制共享。Broadcast Receiver, Activitie,和Service都在应用的manifest中声明。如果你的IPC机制不打算给其他应用使用，设置 `android:exported` 属性为false。这对于同一个UID内包含多个进程的应用，或者在开发后期决定不想通过IPC暴露功能并且不想重写代码的时候非常有用。

如果你的IPC打算让别的应用访问，你可以通过使用Permission标记设置一个安全策略。如果IPC是使用同一个密钥签名的独立的应用间的，使用signature更好一些。

使用Intent

Intent是Android中异步IPC机制的首选。根据你应用的需求，你也许使用sendBroadcast()，sendOrderedBroadcast()或者直接的intent来指定一个应用组件。

注意，有序广播可以被Receiver接收，所以他们也许不会被发送到所有的应用中。 如果你要发送一个intent给指定的Receiver，这个intent必须被直接的发送给这个Receiver。

Intent的发送者能在发送的时候验证Receiver是否有一个许可指定了一个non-Null Permission。只有有那个许可的应用才会收到这个intent。如果广播intent内的数据是敏感的，你应该考虑使用许可来保证恶意应用没有恰当的许可无法注册接收那些消息。这种情况下，可以考虑直接执行这个Receiver而不是发起一个广播。

> 注意：Intent过滤器不能作为安全特性--组件可被intent显式调用，可能会没有符合intent过滤器的数据。你应该在Intent Receiver内执行输入验证，确认对于调用Receiver，Service、或Activity来说格式正确合理。

使用服务

Service经常被用于为其他应用提供服务。每个service类必须在它的manifest文件进行相应的声明。

默认情况下，Service不能被导出和被其他应用执行。如果你加入了任何Intent过滤器到服务的声明中，那么它默认为可以被导出。最好明确声明android:exported元素来确定它按照你设想的运行。可以使用android:permission保护Service。这样做，其他应用在他们自己的manifest文件中将需要声明相应的元素来启动、停止或者绑定到这个Service上。

一个Service可以使用许可保护单独的IPC调用，在执行调用前通过调用checkCallingPermission()来实现。我们建议使用manifest中声明的许可，因为那些是不容易监管的。

使用binder和messenger接口

在Android中，Binders和Messenger是RPC风格IPC的首选机制。必要的话，他们提供一个定义明确的接口，促进彼此的端点认证。

我们强烈鼓励在一定程度上，设计不要求指定许可检查的接口。Binder和Messenger不在应用的manifest中声明，因此你不能直接在Binder上应用声明的许可。它们在应用的manifest中继承许可声明，Service或者Activity内实现了许可。如果你打算创建一个接口，在一个指定binder接口上要求认证和/或者访问控制，这些控制必须在Binder和Messenger的接口中明确添加代码。

如果提供一个需要访问控制的接口，使用checkCallingPermission()来验证调用者是否拥有必要的许可。由于你的应用的id已经被传递到别的接口，因此代表调用者访问一个Service之前这尤其重要。如果调用一个Service提供的接口，如果你没有对给定的Service访问许可，bindService()请求也许会失败。如果调用你自己的应用提供的本地接口，使用clearCallingIdentity()来进行内部安全检查是有用的。

更多关于用服务运行IPC的信息，参见Bound Services

利用BroadcastReceiver

Broadcast receivers是用来处理通过intent发起的异步请求。

默认情况下，Receiver是导出的，并且可以被任何其他应用执行。如果你的BroadcastReceiver打算让其他应用使用，你也许想在应用的manifest文件中使用元素对receiver使用安全许可。这将阻止没有恰当许可的应用发送intent给这个BroadcastReceiver。

动态加载代码

我们不鼓励从应用文件外加载代码。考虑到代码注入或者代码篡改，这样做显著增加了应用暴露的可能，同时也增加了版本管理和应用测试的复杂性。最终可能造成无法验证应用的行为，因此在某些环境下应该被限制。

如果你的应用确实动态加载了代码，最重要的事情是记住运行动态加载的代码与应用具有相同的安全许可。用户决定安装你的应用是基于你的id，他们期望你提供任何运行在应用内部的代码，包括动态加载的代码。

动态加载代码主要的风险在于代码来源于可确认的源头。 如果这个模块是之间直接包含在你的应用中，那么它们不能被其他应用修改，不论代码是本地库或者是使用DexClassLoader加载的类这都是事实。我们见过很多应用实例尝试从不安全的地方加载代码，比如从网络上通过非加密的协议或者从全局可写的位置（比如外部存储）下载数据。这些地方会允许网络上其他人在传输过程中修改其内容，或者允许用户设备上的其他应用修改其内容。

在虚拟机器安全性

Dalvik是安卓的运行时虚拟机(VM)。Dalvik是特别为安卓建立的，但许多其他虚拟机相关的安全代码的也适用于安卓。一般来说，你不应该关心与自己有关的虚拟机的安全问题。你的应用程序在一个安全的沙盒环境下运行，所以系统上的其他进程无法访问你的代码或私人数据。

如果你想更深入了解虚拟机的安全问题，我们建议您熟悉一些现有文献的主题。推荐两个比较流行的资源：

- http://www.securingjava.com/toc.html
- https://www.owasp.org/index.php/Java_Security_Resources

这个文档集中于安卓与其他VM环境不同地方。对于有在其他环境下有VM编程经验开发者来说，这里有两个普遍的问题可能对于编写Android应用来说有些不同：

- 一些虚拟机，比如JVM或者.Net，担任一个安全的边界作用，代码与底层操作系统隔离。在Android上，Dalvik VM不是一个安全边界：应用沙箱是在系统级别实现的，所以Dalvik可以在同一个应用与native代码相互操作，没有任何安全约束。
- 已知的手机上的存储限制，对来发者来说，想要建立模块化应用和使用动态类加载是很常见的。要这么做的时候需要考虑两个资源：一是在哪里恢复你的应用逻辑，二是在哪里存储它们。不要从未验证的资源使用动态类加载器，比如不安全的网络资源或者外部存储，因为那些代码可能被修改为包含恶意行为。

本地代码的安全

一般来说，对于大多数应用开发，我们鼓励开发者使用Android SDK而不是使用[Android NDK]（http://developer.android.com/tools/sdk/ndk/index.html) 的native代码。编译native代码的应用更为复杂，移植性差，更容易包含常见的内存崩溃错误，比如缓冲区溢出。

Android使用Linux内核编译并且与Linux开发相似，如果你打算使用native代码，安全策略尤其有用。与Linux有关的安全问题超出了本文的讨论范围，但读者可以参考Secure Programming for Linux and Unix HOWTO。

与大多数Linux环境的一个重要区别是应用沙箱。在Android中，所有的应用运行在应用沙箱中，包括用native代码编写的应用。在最基本的级别中，与Linux相似，对于开发者来说最好的方式是知道每个应用被分配一个权限非常有限的唯一UID。这里讨论的比Android Security Overview中更细节化，你应该熟悉应用许可，即使你使用的是native代码。

使用HTTPS与SSL

> 编写:craftsmanBai - http://z1ng.net - 原文:
> http://developer.android.com/training/articles/security-ssl.html

SSL，安全套接层(TSL)，是一个常见的用来加密客户端和服务器通信的模块。 但是应用程序错误地使用SSL可能会导致应用程序的数据在网络中被恶意攻击者拦截。为了确保这种情况不在我们的应用中发生，这篇文章主要说明使用网络安全协议常见的陷阱和使用Public-Key Infrastructure(PKI)时一些值得关注的问题。

概念

一个典型的SSL使用场景是，服务器配置中包含了一个证书，有匹配的公钥和私钥。作为SSL客户端和服务端握手的一部分，服务端通过使用public-key cryptography(公钥加密算法)进行证书签名来证明它有私钥。

然而，任何人都可以生成他们自己的证书和私钥，因此一次简单的握手不能证明服务端具有匹配证书公钥的私钥。一种解决这个问题的方法是让客户端拥有一套或者更多可信赖的证书。如果服务端提供的证书不在其中，那么它将不能得到客户端的信任。

这种简单的方法有一些缺陷。服务端应该根据时间升级到强壮的密钥(key rotation)，更新证书中的公钥。不幸的是，现在客户端应用需要根据服务端配置的变化来进行更新。如果服务端不在应用程序开发者的控制下，问题将变得更加麻烦，比如它是一个第三方网络服务。如果程序需要和任意的服务器进行对话，例如web浏览器或者email应用，这种方法也会带来问题。

为了解决这个问题，服务端通常配置了知名的的发行者证书(称为Certificate Authorities(CAs)。提供的平台通常包含了一系列知名可信赖的CAs。Android4.2(Jelly Bean)包含了超过100CAs并在每个发行版中更新。和服务端相似的是，一个CA拥有一个证书和一个私钥。当为一个服务端发布颁发证书的时候，CA用它的私钥为服务端签名。客户端可以通过服务端拥有被已知平台CA签名的证书来确认服务端。

然而，使用CAs又带来了其他的问题。因为CA为许多服务端证书签名，你仍然需要其他的方法来确保你对话的是你想要的服务器。为了解决这个问题，使用CA签名的的证书通过特殊的名字如 gmail.com 或者带有通配符的域名如 *.google.com 来确认服务端。 下面这个例子会使这些概念具体化一些。openssl工具的客户端命令关注Wikipedia服务端证书信息。端口为443（默认为HTTPS）。这条命令将open s_client的输出发送给openssl x509，根据X.509 standard格式化证书中的内容。特别的是，这条命令需要对象（subject），包含服务端名字和签发者（issuer）来确认CA。

```
$ openssl s_client -connect wikipedia.org:443 | openssl x509 -noout -subject -issuer
subject= /serialNumber=sOrr2rKpMVP70Z6E9BT5reY008SJEdYv/C=US/O=*.wikipedia.org/OU=GT03
314600/OU=See www.rapidssl.com/resources/cps (c)11/OU=Domain Control Validated - Rapid
SSL(R)/CN=*.wikipedia.org
issuer= /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
```

可以看到由RapidSSL CA颁发给匹配*.wikipedia.org的服务端证书。

一个HTTP的例子

假设我们有一个知名CA颁发证书的web服务器，那么可以使用下面的代码发送一个安全请求:

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

是的，它就是这么简单。如果我们想要修改HTTP的请求，可以把它交付给HttpURLConnection。Android关于HttpURLConnetcion文档中还有更贴切的关于怎样去处理请求、响应头、posting的内容、cookies管理、使用代理、获取responses等例子。但是就这些确认证书和域名的细节而言，Android框架已经通过API为我们考虑到了这些细节。下面是其他需要关注的问题。

服务器普通问题的验证

假设没有从getInputStream()收到内容，而是抛出了一个异常：

```
javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Tr
ust anchor for certification path not found.
        at org.apache.harmony.xnet.provider.jsse.OpenSSLSocketImpl.startHandshake(Open
SSLSocketImpl.java:374)
        at libcore.net.http.HttpConnection.setupSecureSocket(HttpConnection.java:209)
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.makeSslConnection(Https
URLConnectionImpl.java:478)
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnect
ionImpl.java:433)
        at libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.ja
va:282)
        at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl
.java:177)
        at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionIm
pl.java:271)
```

这种情况发生的原因包括：

1.颁布证书给服务器的CA不是知名的。

2.服务器证书不是CA签名的而是自己签名的。

3.服务器配置缺失了中间CA

下面将会分别讨论当我们和服务器安全连接时如何去解决这些问题。

无法识别证书机构

在这种情况中，SSLHandshakeException异常产生的原因是我们有一个不被系统信任的CA。可能是我们的证书来源于新CA而不被安卓信任，也可能是应用运行版本较老没有CA。更多的时候，一个CA不知名是因为它不是公开的CA，而是政府，公司，教育机构等组织私有的。

幸运的是，我们可以让HttpsURLConnection学会信任特殊的CA。过程可能会让人感到有一些费解，下面这个例子是从InputStream中获得特殊的CA，使用它去创建一个密钥库，用来创建和初始化TrustManager。TrustManager是系统用来验证服务器证书的，这些证书通过使用TrustManager信任的CA和密钥库中的密钥创建。 给定一个新的TrustManager，下面这个例子初始化了一个新的SSLContext，提供了一个SSLSocketFactory，我们可以覆盖来自HttpsURLConnection的默认SSLSocketFactory。这样连接时会使用我们的CA来进行证书验证。

下面是一个华盛顿的大学的组织性的CA的使用例子

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInputStream or ...)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
} finally {
    caInput.close();
}

// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);

// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://certs.cac.washington.edu/CAtest/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

使用一个常用的了解你CA的TrustManager，系统可以确认你的服务器证书来自于一个可信任的发行者。

> 注意：许多网站会提供一个可选解决方案：即让用户安装一个无用的TrustManager。如果你这样做还不如不加密通讯过程，因为任何人都可以在公共wifi热点下，使用伪装成你的服务器的代理发送你的用户流量，进行DNS欺骗，来攻击你的用户。然后攻击者便可记录用户密码和其他个人资料。这种方式可以奏效的原因是因为攻击者可以生成一个证书，并且缺少可以验证该证书是否来自受信任的来源的TrustManager。你的应用可以同任何人会话。所以不要这样做，即使是暂时性的也不行。除非你能始终让你的应用信任服务器证书的签发者。

自签名服务器证书

第二种SSLHandshakeException取决于自签名证书，意味着服务器就是它自己的CA。这同未知证书权威机构类似，因此你同样可以用前面部分中提到的方法。

你可以创建自己的TrustManager，这一次直接信任服务器证书。将应用于证书直接捆绑会有一些缺点，不过我们依然可以确保其安全性。我们应该小心确保我们的自签名证书拥有合适的强密钥。到2012年，一个65537指数位且一年到期的2048位RSA签名是合理的。当轮换密钥时，我们应该查看权威机构（比如NIST）的建议（recommendation）来了解哪种密钥是合适的。

缺少中间证书颁发机构

第三种SSLHandshakeException情况的产生于缺少中间CA。大多数公开的CA不直接给服务器签名。相反，他们使用它们主要的机构（简称根认证机构）证书来给中间认证机构签名，根认证机构这样做，可以离线存储减少危险。然而，像安卓等操作系统通常只直接信任根认证机构，在服务器证书（由中间证书颁发机构签名）和证书验证者（只知道根认证机构）之间留下了一个缺口。为了解决这个问题，服务器并不在SSL握手的过程中只向客户端发送它的证书，而是一系列的从服务器到必经的任何中间机构到达根认证机构的证书。

下面是一个 mail.google.com证书链，以openssls_client命令显示：

```
$ openssl s_client -connect mail.google.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=mail.google.com
   i:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
 1 s:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
   i:/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
---
```

这里显示了一台服务器发送了一个由Thawte SGC CA为mail.google.com颁发的证书，Thawte SGC CA是一个中间证书颁发机构，Thawte SGC CA的证书由被安卓信任的Verisign CA颁发。 然而，配置一台服务器不包括中间证书机构是不常见的。例如，一台服务器导致安卓浏览器的错误和应用的异常:

```
$ openssl s_client -connect egov.uscis.gov:443
---
Certificate chain
 0 s:/C=US/ST=District Of Columbia/L=Washington/O=U.S. Department of Homeland Security
/OU=United States Citizenship and Immigration Services/OU=Terms of use at www.verisign
.com/rpa (c)05/CN=egov.uscis.gov
   i:/C=US/O=VeriSign, Inc./OU=VeriSign Trust Network/OU=Terms of use at https://www.v
erisign.com/rpa (c)10/CN=VeriSign Class 3 International Server CA - G3
---
```

更有趣的是，用大多数桌面浏览器访问这台服务器不会导致类似于完全未知CA的或者自签名的服务器证书导致的错误。这是因为大多数桌面浏览器缓存随着时间的推移信任中间证书机构。一旦浏览器访问并且从一个网站了解到的一个中间证书机构，下一次它将不需要中间证书机构包含证书链。

一些站点会有意让用来提供资源服务的二级服务器像上述所述的那样。比如，他们可能会让他们的主HTML页面用一台拥有全部证书链的服务器来提供，但是像图片，CSS，或者JavaScript等这样的资源用不包含CA的服务器来提供，以此节省带宽。不幸的是，有时这些服务器可能会提供一个在应用中调用的web服务。 这里有两种解决这些问题的方法：

- 配置服务器使它包含服务器链中的中间证书颁发机构
- 或者，像对待不知名的CA一样对待中间CA，并且创建一个TrustManager来直接信任它，就像在前两节中做的那样。

验证主机名常见问题

就像在文章开头提到的那样，有两个关键的部分来确认SSL的连接。第一个是确认证书来源于信任的源，这也是前一个部分关注的焦点。这一部分关注第二部分：确保你当前对话的服务器有正确的证书。当情况不是这样时，你可能会看到这样的典型错误：

```
java.io.IOException: Hostname 'example.com' was not verified
        at libcore.net.http.HttpConnection.verifySecureSocketHostname(HttpConnection.j
ava:223)
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnect
ionImpl.java:446)
        at libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.ja
va:282)
        at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl
.java:177)
        at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionIm
pl.java:271)
```

服务器配置错误可能会导致这种情况发生。服务器配置了一个证书，这个证书没有匹配的你想连接的服务器的subject或者subject可选的命名域。一个证书被许多不同的服务器使用是可能的。比如，使用 openssl s_client -connect google.com:443 |openssl x509 -text 查看google证书，你可以看到一个subject支持 *google.con* .youtube.com, *.android.com或者其他的。这种错误只会发生在你所连接的服务器名称没有被证书列为可接受。

不幸的是另外一种原因也会导致这种情况发生：虚拟化服务。当用HTTP同时拥有一个以上主机名的服务器共享时，web服务器可以从 HTTP/1.1请求中找到客户端需要的目标主机名。不行的是，使用HTTPS会使情况变得复杂，因为服务器必须知道在发现HTTP请求前返回哪一个证书。为了解决这个问题，新版本的SSL，特别是TLSV.1.0和之后的版本，支持服务器名指示(SNI)，允许SSL客户端为服务端指定目标主机名，从而返回正确的证书。 幸运的是，从安卓2.3开始，HttpsURLConnection支持SNI。不幸的是，Apache HTTP客户端不这样，这也是我们不鼓励用它的原因之一。如果你需要支持安卓2.2或者更老的版本或者Apache HTTP客户端，一个解决方法是建立一个可选的虚拟化服务并且使用特别的端口，这样服务端就能够清楚该返回哪一个证书。

采用不使用你的虚拟服务的主机名HostnameVerifier而不是服务器默认的来替换，是很重要的选择。

注意：替换HostnameVerifier可能会非常危险，如果另外一个虚拟服务不在你的控制下，中间人攻击可能会直接使流量到达另外一台服务器而超出你的预想。 如果你仍然确定你想覆盖主机名验证，这里有一个为单URLConnection替换验证过程的例子：

```
// Create an HostnameVerifier that hardwires the expected hostname.
// Note that is different than the URL's hostname:
// example.com versus example.org
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        HostnameVerifier hv =
            HttpsURLConnection.getDefaultHostnameVerifier();
        return hv.verify("example.com", session);
    }
};

// Tell the URLConnection to use our HostnameVerifier
URL url = new URL("https://example.org/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setHostnameVerifier(hostnameVerifier);
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

但是请记住，如果你发现你在替换主机名验证，特别是虚拟服务，另外一个虚拟主机不在你的控制的情况是非常危险的，你应该找到一个避免这种问题产生的托管管理。

关于直接使用SSL Socket的警告

到目前为止，这些例子聚焦于使用HttpsURLConnection上。有时一些应用需要让SSL和HTTP分开。举个例子，一个email应用可能会使用SSL的变种，SMTP,POP3,IMAP等。在那些例子中，应用程序会想使用SSLSocket直接连接，与HttpsURLConnection做的方法相似。 这种技术到目前为止处理了证书验证问题，也应用于SSLSocket中。事实上，当使用常规的TrustManager时，传递给HttpsURLConnection的是SSLSocketFactory。如果你需要一个带常规的SSLSocket的TrustManager，采取下面的步骤使用SSLSocketFactory来创建你的SSLSocket。

> 注意： SSLSocket不具有主机名验证功能。它取决于它自己的主机名验证，通过传入预期的主机名调用getDefaultHostNameVerifier()。进一步需要注意的是，当发生错误时，HostnameVerifier.verify()不知道抛出异常，而是返回一个布尔值，你需要进一步明确的检查。 下面是一个演示的方法。这个例子演示了当它连接gmail.com 443端口并且没有SNI支持的时候，你将会收到一个mail.google.com的证书。你需要确保证书的确是mail.google.com的。

```
// Open SSLSocket directly to gmail.com
SocketFactory sf = SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) sf.createSocket("gmail.com", 443);
HostnameVerifier hv = HttpsURLConnection.getDefaultHostnameVerifier();
SSLSession s = socket.getSession();

// Verify that the certicate hostname is for mail.google.com
// This is due to lack of SNI support in the current SSLSocket.
if (!hv.verify("mail.google.com", s)) {
    throw new SSLHandshakeException("Expected mail.google.com, "
                                    "found " + s.getPeerPrincipal());
}

// At this point SSLSocket performed certificate verificaiton and
// we have performed hostname verification, so it is safe to proceed.

// ... use socket ...
socket.close();
```

黑名单

SSL 主要依靠CA来确认证书来自正确无误服务器和域名的所有者。少数情况下，CA被欺骗，或者在Comodo和DigiNotar的例子中，一个主机名的证书被颁发给了除了服务器和域名的拥有者之外的人，导致被破坏。

为了减少这种危险，安卓可以将一些黑名单或者整个CA列入黑名单。尽管名单是以前是嵌入操作系统的，从安卓4.2开始，这个名单在以后的方案中可以远程更新。

阻塞

一个应用可以通过阻塞技术保护它自己免于受虚假证书的欺骗。这是简单运用使用未知CA的例子，限制应用信任的CA仅来自被应用使用的服务器。阻止了来自系统中另外一百多个CA的欺骗而导致的应用安全通道的破坏。

客户端验证

这篇文章聚焦在SSL的使用者同服务器的安全对话上。SSL也支持服务端通过验证客户端的证书来确认客户端的身份。这种技术也与TrustManager的特性相似。可以参考在HttpsURLConnection文档中关于创建一个常规的KeyManager的讨论。

nogotofail：网络流量安全测试工具

对于已知的TLS／SSL漏洞和错误，nogotofail提供了一个简单的方法来确认你的应用程序是安全的。它是一个自动化的、强大的、用于测试网络的安全问题可扩展性的工具，任何设备的网络流量都可以通过它。 nogotofail主要应用于三种场景：

- 发现错误和漏洞。
- 验证修补程序和等待回归。
- 了解应用程序和设备产生的交通。

nogotofail 可以工作在Android，iOS，Linux，Windows，Chrome OS，OSX环境下，事实上任何需要连接到Internet的设备都可以。Android和Linux环境下有简单易用获取通知的客户端配置设置，以及本身可以作为靶机，部署为一个路由器，VPN服务器，或代理。 你可以在nogotofail开源项目访问该工具。

更新你的Security Provider来对抗SSL漏洞利用

> 编写:craftsmanBai - http://z1ng.net - 原文: http://developer.android.com/training/articles/security-gms-provider.html

安卓依靠security provider保障网络通信安全。然而有时默认的security provider存在安全漏洞。为了防止这些漏洞被利用，Google Play services 提供了一个自动更新设备的security provider的方法来对抗已知的漏洞。通过调用Google Play services方法，可以确保你的应用运行在可以抵抗已知漏洞的设备上。

举个例子，OpenSSL的漏洞(CVE-2014-0224)会导致中间人攻击，在通信双方不知情的情况下解密流量。Google Play services 5.0提供了一个补丁，但是必须确保应用安装了这个补丁。通过调用Google Play services方法，可以确保你的应用运行在可抵抗攻击的安全设备上。

注意：更新设备的security provider不是更新android.net.SSLCertificateSocketFactory.比起使用这个类，我们更鼓励应用开发者使用融入密码学的高级方法。大多数应用可以使用类似HttpsURLConnection，HttpClient，AndroidHttpClient这样的API，而不必去设置TrustManager或者创建一个SSLCertificateSocketFactory。

使用ProviderInstaller给Security Provider打补丁

使用providerinstaller类来更新设备的security provider。你可以通过调用该类的方法installIfNeeded()(或者 installifneededasync)来验证security provider是否为最新的(必要的话更新它)。

当你调用installifneeded时，providerinstaller会做以下事情：

- 如果设备的Provider成功更新(或已经是最新的)，该方法返回正常。
- 如果设备的Google Play services 库已经过时了，这个方法抛出googleplayservicesrepairableexception异常表明无法更新Provider。应用程序可以捕获这个异常并向用户弹出合适的对话框提示更新Google Play services。
- 如果产生了不可恢复的错误，该方法抛出googleplayservicesnotavailableexception表示它无法更新Provider。应用程序可以捕获异常并选择合适的行动，如显示标准问题解决流程图。

installifneededasync方法类似，但它不抛出异常，而是通过相应的回调方法，以提示成功或失败。

如果installifneeded需要安装一个新的Provider，可能耗费30-50毫秒（较新的设备）到350毫秒（旧设备）。如果security provider已经是最新的，该方法需要的时间量可以忽略不计。为了避免影响用户体验：

- 线程加载后立即在后台网络线程中调用installifneeded，而不是等待线程尝试使用网络。（多次调用该方法没有害处，如果安全提供程序不需要更新它会立即返回。）
- 如果用户体验会受线程阻塞的影响——比如从UI线程中调用，那么使用installifneededasync()调用该方法的异步版本。（当然，如果你要这样做，在尝试任何安全通信之前必须等待操作完成。providerinstaller调用监听者的onproviderinstalled()方法发出成功信号。

警告：如果providerinstaller无法安装更新Provider，您的设备security provider会容易受到已知漏洞的攻击。你的程序等同于所有HTTP通信未被加密。 一旦Provider更新，所有安全API（包括SSL API）的调用会经过它(但这并不适用于android.net.sslcertificatesocketfactory，面对cve-2014-0224这种漏洞仍然是脆弱的)。

同步修补

修补security provider最简单的方法就是调用同步方法installIfNeeded().如果用户体验不会被线程阻塞影响的话，这种方法很合适。

举个例子，这里有一个sync adapter会更新security provider。由于它运行在后台，因此在等待security provider更新的时候线程阻塞是可以的。sync adapter调用installifneeded()更新security provider。如果返回正常，sync adapter可以确保security provider是最新的。如果返回异常，sync adapter可以采取适当的行动（如提示用户更新Google Play services）。

```
/**
 * Sample sync adapter using {@link ProviderInstaller}.
 */
public class SyncAdapter extends AbstractThreadedSyncAdapter {

  ...

  // This is called each time a sync is attempted; this is okay, since the
  // overhead is negligible if the security provider is up-to-date.
  @Override
  public void onPerformSync(Account account, Bundle extras, String authority,
      ContentProviderClient provider, SyncResult syncResult) {
    try {
      ProviderInstaller.installIfNeeded(getContext());
    } catch (GooglePlayServicesRepairableException e) {

      // Indicates that Google Play services is out of date, disabled, etc.

      // Prompt the user to install/update/enable Google Play services.
      GooglePlayServicesUtil.showErrorNotification(
          e.getConnectionStatusCode(), getContext());

      // Notify the SyncManager that a soft error occurred.
      syncResult.stats.numIOExceptions++;
      return;

    } catch (GooglePlayServicesNotAvailableException e) {
      // Indicates a non-recoverable error; the ProviderInstaller is not able
      // to install an up-to-date Provider.

      // Notify the SyncManager that a hard error occurred.
      syncResult.stats.numAuthExceptions++;
      return;
    }

    // If this is reached, you know that the provider was already up-to-date,
    // or was successfully updated.
  }
}
```

异步修补

更新security provider可能耗费350毫秒（旧设备）。如果在一个会直接影响用户体验的线程中更新，如UI线程，那么你不会希望进行同步更新，因为这可能导致应用程序或设备冻结直到操作完成。因此你应该使用异步方法installifneededasync()。方法通过调用回调函数来反馈其成功或失败。 例如，下面是一些关于更新security provider在UI线程中的活动的代码。调用installifneededasync()来更新security provider，并指定自己为监听器接收成功或失败的通

知。如果security provider是最新的或更新成功，会调用onproviderinstalled()方法，并且知道通信是安全的。如果security provider无法更新，会调用onproviderinstallfailed()方法，并采取适当的行动（如提示用户更新Google Play services）

```
/**
 * Sample activity using {@link ProviderInstaller}.
 */
public class MainActivity extends Activity
    implements ProviderInstaller.ProviderInstallListener {

  private static final int ERROR_DIALOG_REQUEST_CODE = 1;

  private boolean mRetryProviderInstall;

  //Update the security provider when the activity is created.
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ProviderInstaller.installIfNeededAsync(this, this);
  }

  /**
   * This method is only called if the provider is successfully updated
   * (or is already up-to-date).
   */
  @Override
  protected void onProviderInstalled() {
    // Provider is up-to-date, app can make secure network calls.
  }

  /**
   * This method is called if updating fails; the error code indicates
   * whether the error is recoverable.
   */
  @Override
  protected void onProviderInstallFailed(int errorCode, Intent recoveryIntent) {
    if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {
      // Recoverable error. Show a dialog prompting the user to
      // install/update/enable Google Play services.
      GooglePlayServicesUtil.showErrorDialogFragment(
          errorCode,
          this,
          ERROR_DIALOG_REQUEST_CODE,
          new DialogInterface.OnCancelListener() {
            @Override
            public void onCancel(DialogInterface dialog) {
              // The user chose not to take the recovery action
              onProviderInstallerNotAvailable();
            }
          });
    } else {
      // Google Play services is not available.
```

```
        onProviderInstallerNotAvailable();
      }
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {
      super.onActivityResult(requestCode, resultCode, data);
      if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
        // Adding a fragment via GooglePlayServicesUtil.showErrorDialogFragment
        // before the instance state is restored throws an error. So instead,
        // set a flag here, which will cause the fragment to delay until
        // onPostResume.
        mRetryProviderInstall = true;
      }
    }

    /**
     * On resume, check to see if we flagged that we need to reinstall the
     * provider.
     */
    @Override
    protected void onPostResume() {
      super.onPostResult();
      if (mRetryProviderInstall) {
        // We can now safely retry installation.
        ProviderInstall.installIfNeededAsync(this, this);
      }
      mRetryProviderInstall = false;
    }

    private void onProviderInstallerNotAvailable() {
      // This is reached if the provider cannot be updated for some reason.
      // App should consider all HTTP communication to be vulnerable, and take
      // appropriate action.
    }
  }
```

使用设备管理策略增强安全性

> 编写:craftsmanBai - http://z1ng.net - 原文: http://developer.android.com/training/enterprise/device-management-policy.html

Android 2.2(API Level 8)之后，Android平台通过设备管理API提供系统级的设备管理能力。

在这一小节中，你将学到如何通过使用设备管理策略创建安全敏感的应用程序。比如某应用可被配置为：在给用户显示受保护的内容之前，确保已设置一个足够强度的锁屏密码。

定义并声明你的策略

首先，你需要定义多种在功能层面提供支持的策略。这些策略可以包括屏幕锁密码强度、密码过期时间以及加密等等方面。

你须在res/xml/device_admin.xml中声明选择的策略集，它将被应用强制实行。在Android manifest也需要引用声明的策略集。

每个声明的策略对应DevicePolicyManager中一些相关设备的策略方法（例如定义最小密码长度或最少大写字母字符数）。如果一个应用尝试调用XML中没有对应策略的方法，程序在会运行时抛出一个SecurityException异常。

如果应用程序试图管理其他策略，那么强制锁force-lock之类的其他权限就会发挥作用。正如你将看到的，作为设备管理权限激活过程的一部分，声明策略的列表会在系统屏幕上显示给用户。 如下代码片段在res/xml/device_admin.xml中声明了密码限制策略：

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies>
        <limit-password />
    </uses-policies>
</device-admin>
```

在Android manifest引用XML策略声明：

```
<receiver android:name=".Policy$PolicyAdmin"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
        android:resource="@xml/device_admin" />
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

创建一个设备管理接受端

创建一个设备管理广播接收端（broadcast receiver），可以接收到与你声明的策略有关的事件通知。也可以对应用程序有选择地重写回调函数。

在同样的应用程序（Device Admin）中，当设备管理（device administrator）权限被用户设为禁用时，已配置好的策略就会从共享偏好设置（shared preference）中擦除。

你应该考虑实现与你的应用业务逻辑相关的策略。例如，你的应用可以采取一些措施来降低安全风险，如：删除设备上的敏感数据，禁用远程同步，对管理员的通知提醒等等。

为了让广播接收端能够正常工作，请务必在Android manifest中注册下面代码片段所示内容。

```
<receiver android:name=".Policy$PolicyAdmin"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
        android:resource="@xml/device_admin" />
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

激活设备管理器

在执行任何策略之前，用户需要手动将程序激活为具有设备管理权限，下面的程序片段显示了如何触发设置框以便让用户为你的程序激活权限。

通过指定EXTRA_ADD_EXPLANATION给出明确的说明信息，以告知用户为应用程序激活设备管理权限的好处。

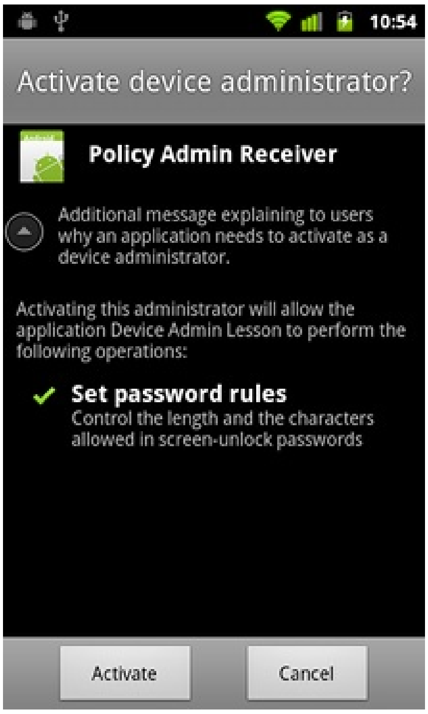
```
if (!mPolicy.isAdminActive()) {

    Intent activateDeviceAdminIntent =
        new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);

    activateDeviceAdminIntent.putExtra(
        DevicePolicyManager.EXTRA_DEVICE_ADMIN,
        mPolicy.getPolicyAdmin());

    // It is good practice to include the optional explanation text to
    // explain to user why the application is requesting to be a device
    // administrator. The system will display this message on the activation
    // screen.
    activateDeviceAdminIntent.putExtra(
        DevicePolicyManager.EXTRA_ADD_EXPLANATION,
        getResources().getString(R.string.device_admin_activation_message));

    startActivityForResult(activateDeviceAdminIntent,
        REQ_ACTIVATE_DEVICE_ADMIN);
}
```



如果用户选择"Activate"，程序就会获取设备管理员权限并可以开始配置和执行策略。 当然，程序也需要做好处理用户选择放弃激活的准备，比如用户点击了“取消”按钮，返回键或者HOME键的情况。因此，如果有必要的话，策略设置中的onResume())方法需要加入重新评估的逻辑判断代码，以便将设备管理激活选项展示给用户。

实施设备策略控制

在设备管理权限成功激活后，程序就会根据请求的策略来配置设备策略管理器。要牢记，新策略会被添加到每个版本的Android中。所以你需要在程序中做好平台版本的检测，以便新策略能被老版本平台很好的支持。例如，“密码中含有的最少大写字符数”这个安全策略只有在高于API Level 11（Honeycomb）的平台才被支持，以下代码则演示了如何在运行时检查版本：

```
DevicePolicyManager mDPM = (DevicePolicyManager)
        context.getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName mPolicyAdmin = new ComponentName(context, PolicyAdmin.class);
...
mDPM.setPasswordQuality(mPolicyAdmin, PASSWORD_QUALITY_VALUES[mPasswordQuality]);
mDPM.setPasswordMinimumLength(mPolicyAdmin, mPasswordLength);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    mDPM.setPasswordMinimumUpperCase(mPolicyAdmin, mPasswordMinUpperCase);
}
```

这样程序就可以执行策略了。当程序无法访问正确的锁屏密码的时候，通过设备策略管理器（Device Policy Manager）API可以判断当前密码是否适用于请求的策略。如果当前锁屏密码满足策略，设备管理API不会采取纠正措施。明确地启动设置程序中的系统密码更改界面是应用程序的责任。例如：

```
if (!mDPM.isActivePasswordSufficient()) {
    ...
    // Triggers password change screen in Settings.
    Intent intent =
        new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
    startActivity(intent);
}
```

一般来说，用户可以从可用的锁屏机制中任选一个，例如“无”、“图案”、“PIN码”（数字）或密码（字母数字）。当一个密码策略配置好后，那些比已定义密码策略弱的密码会被禁用。比如，如果配置了密码级别为“Numeric”，那么用户只可以选择PIN码（数字）或者密码（字母数字）。

一旦设备通过设置适当的锁屏密码处于被保护的状态，应用程序便允许访问受保护的内容。

```
if (!mDPM.isAdminActive(..)) {
    // Activates device administrator.
    ...
} else if (!mDPM.isActivePasswordSufficient()) {
    // Launches password set-up screen in Settings.
    ...
} else {
    // Grants access to secure content.
    ...
    startActivity(new Intent(context, SecureActivity.class));
}
```

测试程序

> 编写:kesenhoo - 原文:http://developer.android.com/training/testing.html

These classes and articles provide information about how to test your Android application.

Testing Your Activity

How to test Activities in your Android applications.

测试你的Activity

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/index.html

我们应该把编写和运行测试作为Android应用开发周期的一部分。完备的测试可以帮助我们在开发过程中尽早发现漏洞，并让我们对自己的代码更有信心。

测试用例定义了一系列对象和方法从而独立进行多个测试。测试用例可以编写成测试组并按计划的运行，由测试框架组织成一个可以重复运行的测试Runner（运行器，译者注）。

这节内容将会讲解如何基于最流行的JUnit框架来自定义测试框架。我们可以编写测试用例来测试我们应用程序的特定行为，并在不同的Android设备上检测一致性。测试用例还可以用来描述应用组件的预期行为，并作为内部代码文档。

课程

- 建立测试环境

学习如何创建测试项目

- 创建与执行测试用例

学习如何写测试用例来检验Activity中的特性，并使用Android框架提供的Instrumentation运行用例。

- 测试**UI**组件

学习如何编写UI测试用例

- 创建单元测试

学习如何隔离开Activity执行单元测试

- 创建功能测试

学习如何执行功能测试来检验各Activity之间的交互

建立测试环境

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/preparing-activity-testing.html

在开始编写并运行我们的测试之前，我们应该建立测试开发环境。本小节将会讲解如何建立Eclipse IDE来构建和运行我们的测试，以及怎样用Gradle构建工具在命令行下构建和运行我们的测试。

> 注意: 本小节基于的是Eclipse及ADT插件。然而，你在自己测试开发时可以自由选用IDE或命令行。

用Eclipse建立测试

安装了Android Developer Tools (ADT) 插件的Eclipse将为我们创建，构建，以及运行Android程序提供一个基于图形界面的集成开发环境。Eclipse可以自动为我们的Android应用项目创建一个对应的测试项目。

开始在Eclipse中创建测试环境:

1. 如果还没安装Eclipse ADT插件，请先下载安装。
2. 导入或创建我们想要测试的Android应用项目。
3. 生成一个对应于应用程序项目测试的测试项目。为导入项目生成一个测试项目: a.在项目浏览器里，右击我们的应用项目，然后选择**Android Tools > New Test Project** b.在新建Android测试项目面板，为我们的测试项目设置合适的参数，然后点击**Finish**

现在应该可以在Eclipse环境中创建，构建和运行测试项目了。想要继续学习如何在Eclipse中进行这些任务，可以阅读创建与执行测试用例

用命令行建立测试

如果正在使用Gradle version 1.6或者更高的版本作为构建工具，可以用Gradle Wrapper创建。构建和运行Android应用测试。确保在 `gradle.build` 文件中， `defaultConfig` 部分中的minSdkVersion属性是8或更高。可以参考包含在下载包中的示例文件gradle.build

用Gradle Wrapper运行测试:

1. 连接Android真机或开启Android模拟器。

2. 在项目目录运行如下命令:

> ./gradlew build connectedCheck

进一步学习Gradle关于Android测试的内容，参看Gradle Plugin User Guide。

进一步学习使用Gradle及其它命令行工具，参看Testing from Other IDEs.。

本节示例代码AndroidTestingFun.zip

创建与执行测试用例

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/activity-basic-testing.html

为了验证应用的布局设计和功能是否符合预期，为应用的每个Activity建立测试非常重要。对于每一个测试，我们需要在测试用例中创建一个个独立的部分，包括测试数据，前提条件和Activity的测试方法。之后我们就可以运行测试并得到测试报告。如果有任何测试没有通过，这表明在我们代码中可能有潜在的缺陷。

> 注意: 在测试驱动开发（TDD）方法中, 不推荐先编写大部分或整个应用，并在开发完成后再运行测试。而是应该先编写测试，然后及时编写正确的代码，以通过测试。通过更新测试案例来反映新的功能需求，并以此反复。

创建一个测试用例

Activity测试都是通过结构化的方式编写的。请务必把测试代码放在一个单独的包内，从而与被测试的代码分开。

按照惯例，测试包的名称应该遵循与应用包名相同的命名方式，在应用包名后接“.tests”。在创建的测试包中，为我们的测试用例添加Java类。按照惯例，测试用例名称也应遵循要测试的Java或Android的类相同的名称，并增加后缀“Test”。

要在Eclipse中创建一个新的测试用例可遵循如下步骤：

a. 在Package Explorer中，右键点击待测试工程的src/文件夹，**New > Package**。

b. 设置文件夹名称`<你的包名称>.tests`（比如, `com.example.android.testingfun.tests`）并点击**Finish**。

c. 右键点击创建的测试包，并选择**New > Calss**。

d. 设置文件名称`<你的Activity名称>Test`（比如, `MyFirstTestActivityTest`），然后点击**Finish**。

建立测试数据集(Fixture)

测试数据集包含运行测试前必须生成的一些对象。要建立测试数据集，可以在我们的测试中覆写setUp()和tearDown()方法。测试会在运行任何其它测试方法之前自动执行setUp()方法。我们可以用这些方法使得被测试代码与测试初始化和清理是分开的。

在你的Eclipse中建立测试数据集:

1. 在 Package Explorer中双击测试打开之前编写的测试用例，然后修改测试用例使它继承ActivityTestCase的子类。比如：

```
public class MyFirstTestActivityTest
        extends ActivityInstrumentationTestCase2<MyFirstTestActivity> {
```

2. 下一步，给测试用例添加构造函数和setUp()方法，并为我们想测试的Activity添加变量声明。比如:

```
public class MyFirstTestActivityTest
        extends ActivityInstrumentationTestCase2<MyFirstTestActivity> {

    private MyFirstTestActivity mFirstTestActivity;
    private TextView mFirstTestText;

    public MyFirstTestActivityTest() {
        super(MyFirstTestActivity.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mFirstTestActivity = getActivity();
        mFirstTestText =
                (TextView) mFirstTestActivity
                .findViewById(R.id.my_first_test_text_view);
    }
}
```

构造函数是由测试用的Runner调用，用于初始化测试类的，而setUp()方法是由测试Runner在其他测试方法开始前运行的。

通常在 `setUp()` 方法中，我们应该:

- 为 `setUp()` 调用父类构造函数，这是JUnit要求的。
- 初始化测试数据集的状态，具体而言：
  - 定义保存测试数据及状态的实例变量
  - 创建并保存正在测试的Activity的引用实例。
  - 获得想要测试的Activity中任何UI组件的引用。

我们可以使用getActivity()方法得到正在测试的Activity的引用。

增加一个测试前提

我们最好在执行测试之前，检查测试数据集的设置是否正确，以及我们想要测试的对象是否已经正确地初始化。这样，测试就不会因为有测试数据集的设置错误而失败。按照惯例，验证测试数据集的方法被称为 `testPreconditions()` 。

例如，我们可能想添加一个像这样的 `testPreconditons()` 方法:

```
public void testPreconditions() {
    assertNotNull(“mFirstTestActivity is null”, mFirstTestActivity);
    assertNotNull(“mFirstTestText is null”, mFirstTestText);
}
```

Assertion（断言，译者注）方法源自于JunitAssert类。通常，我们可以使用断言来验证某一特定的条件是否是真的。

- 如果条件为假，断言方法抛出一个AssertionFailedError异常，通常会由测试Runner报告。我们可以在断言失败时给断言方法添加一个字符串作为第一个参数从而给出一些上下文详细信息。
- 如果条件为真，测试通过。

在这两种情况下，Runner都会继续运行其它测试用例的测试方法。

添加一个测试方法来验证Activity

下一步，添加一个或多个测试方法来验证Activity布局和功能。

例如，如果我们的Activity含有一个TextView，可以添加如下方法来检查它是否有正确的标签文本:

```
public void testMyFirstTestTextView_labelText() {
    final String expected =
            mFirstTestActivity.getString(R.string.my_first_test);
    final String actual = mFirstTestText.getText().toString();
    assertEquals(expected, actual);
}
```

该 `testMyFirstTestTextView_labelText()` 方法只是简单的检查Layout中TextView的默认文本是否和 `strings.xml` 资源中定义的文本一样。

> 注意：当命名测试方法时，我们可以使用下划线将被测试的内容与测试用例区分开。这种风格使得我们可以更容易分清哪些是测试用例。

做这种类型的字符串比较时，推荐从资源文件中读取预期字符串，而不是在代码中硬性编写字符串做比较。这可以防止当资源文件中的字符串定义被修改时，会影响到测试的效果。

为了进行比较，预期的和实际的字符串都要做为assertEquals()方法的参数。如果值是不一样的，断言将抛出一个AssertionFailedError异常。

如果添加了一个 `testPreconditions()` 方法，我们可以把测试方法放在testPreconditions之后。

要参看一个完整的测试案例，可以参考本节示例中的MyFirstTestActivityTest.java。

构建和运行测试

我们可以在Eclipse中的包浏览器（Package Explorer）中运行我们的测试。

利用如下步骤构建和运行测试:

1. 连接一个Android设备，在设备或模拟器中，打开设置菜单，选择开发者选项并确保启用USB调试。
2. 在包浏览器(Package Explorer)中，右键单击测试类，并选择**Run As > Android Junit Test**。
3. 在Android设备选择对话框，选择刚才连接的设备，然后单击“确定”。
4. 在JUnit视图，验证测试是否通过,有无错误或失败。

本节示例代码AndroidTestingFun.zip

测试UI组件

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/activity-ui-testing.html

通常情况下，Activity，包括用户界面组件（如按钮，复选框，可编辑的文本域，和选框）允许用户与Android应用程序交互。本节介绍如何对一个简单的带有按钮的界面交互测试。我们可以使用相同的步骤来测试其他更复杂的UI组件。

> 注意: 这一节的测试方法叫做白盒测试，因为我们拥有要测试应用程序的源码。Android Instrumentation框架适用于创建应用程序中UI部件的白盒测试。用户界面测试的另一种类型是黑盒测试，即无法得知应用程序源代码的类型。这种类型的测试可以用来测试应用程序如何与其他应用程序，或与系统进行交互。黑盒测试不包括在本节中。了解更多关于如何在你的Android应用程序进行黑盒测试，请阅读UI Testing guide。

要参看完整的测试案例，可以查看本节示例代码中的 `ClickFunActivityTest.java` 文件。

使用 Instrumentation 建立UI测试

当测试拥有UI的Activity时，被测试的Activity在UI线程中运行。然而，测试程序会在程序自己的进程中，单独的一个线程内运行。这意味着，我们的测试程序可以获得UI线程的对象，但是如果它尝试改变UI线程对象的值，会得到 `WrongThreadException` 错误。

为了安全地将 `Intent` 注入到 `Activity` ，或是在UI线程中执行测试方法，我们可以让测试类继承于ActivityInstrumentationTestCase2。要学习如何在UI线程运行测试方法，请看在UI线程测试。

建立测试数据集（Fixture）

当为UI测试建立测试数据集时，我们应该在setUp()方法中指定touch mode。把touch mode设置为真可以防止在执行编写的测试方法时，人为的UI操作获取到控件的焦点（比如,一个按钮会触发它的点击监听器）。确保在调用getActivity()方法前调用了setActivityInitialTouchMode)。

比如:

```
public class ClickFunActivityTest
        extends ActivityInstrumentationTestCase2 {
    ...
    @Override
    protected void setUp() throws Exception {
        super.setUp();

        setActivityInitialTouchMode(true);

        mClickFunActivity = getActivity();
        mClickMeButton = (Button)
                mClickFunActivity
                .findViewById(R.id.launch_next_activity_button);
        mInfoTextView = (TextView)
                mClickFunActivity.findViewById(R.id.info_text_view);
    }
}
```

添加测试方法确认UI响应表现

UI测试目标应包括:

. 检验*Activity*启动时*Button*在正确布局位置显示。 . 检验TextView初始化时是隐藏的。 *. 检验TextView在Button点击时显示预期的字符串

接下来的部分会演示怎样实现上述验证方法

验证Button布局参数

我们应该像如下添加的测试方法那样。验证Activity中的按钮是否正确显示:

```
@MediumTest
public void testClickMeButton_layout() {
    final View decorView = mClickFunActivity.getWindow().getDecorView();

    ViewAsserts.assertOnScreen(decorView, mClickMeButton);

    final ViewGroup.LayoutParams layoutParams =
            mClickMeButton.getLayoutParams();
    assertNotNull(layoutParams);
    assertEquals(layoutParams.width, WindowManager.LayoutParams.MATCH_PARENT);
    assertEquals(layoutParams.height, WindowManager.LayoutParams.WRAP_CONTENT);
}
```

在调用assertOnScreen()方法时，传递根视图以及期望呈现在屏幕上的视图作为参数。如果想呈现的视图没有在根视图中,该方法会抛出一个AssertionFailedError异常，否则测试通过。

我们也可以通过获取一个ViewGroup.LayoutParams对象的引用验证Button布局是否正确，然后调用 `assert` 方法验证Button对象的宽高属性值是否与预期值一致。

`@MediumTest` 注解指定测试是如何归类的（和它的执行时间相关）。要了解更多有关测试的注解，见本节示例。

验证TextView的布局参数

可以像这样添加一个测试方法来验证TextView最初是隐藏在Activity中的:

```
@MediumTest
public void testInfoTextView_layout() {
    final View decorView = mClickFunActivity.getWindow().getDecorView();
    ViewAsserts.assertOnScreen(decorView, mInfoTextView);
    assertTrue(View.GONE == mInfoTextView.getVisibility());
}
```

我们可以调用 `getDecorView()` 方法得到一个Activity中修饰试图（Decor View）的引用。要修饰的View在布局层次视图中是最上层的ViewGroup(FrameLayout)

验证按钮的行为

可以使用如下测试方法来验证当按下按钮时TextView变得可见:

```
@MediumTest
public void testClickMeButton_clickButtonAndExpectInfoText() {
    String expectedInfoText = mClickFunActivity.getString(R.string.info_text);
    TouchUtils.clickView(this, mClickMeButton);
    assertTrue(View.VISIBLE == mInfoTextView.getVisibility());
    assertEquals(expectedInfoText, mInfoTextView.getText());
}
```

在测试中调用clickView()可以让我们用编程方式点击一个按钮。我们必须传递正在运行的测试用例的一个引用和要操作按钮的引用。

> 注意:TouchUtils辅助类提供与应用程序交互的方法可以方便进行模拟触摸操作。我们可以使用这些方法来模拟点击，轻敲，或应用程序屏幕拖动View。
>
> 警告TouchUtils方法的目的是将事件安全地从测试线程发送到UI线程。我们不可以直接在UI线程或任何标注@UIThread的测试方法中使用TouchUtils这样做可能会增加错误线程异常。

应用测试注解

@SmallTest

```
标志该测试方法是小型测试的一部分。
```

@MediumTest

```
标志该测试方法是中等测试的一部分。
```

@LargeTest

```
标志该测试方法是大型测试的一部分。
```

通常情况下，如果测试方法只需要几毫秒的时间，那么它应该被标记为@SmallTest，长时间运行的测试（100毫秒或更多）通常被标记为@MediumTest或@LargeTest，这主要取决于测试访问资源在网络上或在本地系统。 可以参看Android Tools Protip，它可以更好地指导我们使用测试注释

我们可以创建其它的测试注释来控制测试的组织和运行。要了解更多关于其他注释的信息，见Annotation类参考。

本节示例代码AndroidTestingFun.zip

创建单元测试

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/activity-unit-testing.html

Activity单元测试可以快速且独立地（和系统其它部分分离）验证一个Activity的状态以及其与其它组件交互的正确性。一个单元测试通常用来测试代码中最小单位的代码块（可以是一个方法，类，或者组件），而且也不依赖于系统或网络资源。比如说，你可以写一个单元测试去检查Activity是否正确地布局或者是否可以正确地触发一个Intent对象。

单元测试一般不适合测试与系统有复杂交互的UI。我们应该使用如同测试UI组件所描述的 `ActivityInstrumentationTestCase2` 来对这类UI交互进行测试。

这节内容将会讲解如何编写一个单元测试来验证一个Intent是否正确地触发了另一个Activity。由于测试是与环境独立的，所以Intent实际上并没有发送给Android系统，但我们可以检查Intent对象的载荷数据是否正确。读者可以参考一下示例代码中的 `LaunchActivityTest.java` ，将它作为一个例子，了解完备的测试用例是怎么样的。

> 注意: 如果要针对系统或者外部依赖进行测试，我们可以使用Mocking Framework的Mock类，并把它集成到我们的你的单元测试中。要了解更多关于Android提供的Mocking Framework内容请参考Mock Object Classes。

编写一个Android单元测试例子

ActiviUnitTestCase类提供对于单个Activity进行分离测试的支持。要创建单元测试，我们的测试类应该继承自 `ActiviUnitTestCase` 。继承 `ActiviUnitTestCase` 的Activity不会被Android自动启动。要单独启动Activity，我们需要显式的调用startActivity()方法，并传递一个Intent来启动我们的目标Activity。

例如：

```
public class LaunchActivityTest
        extends ActivityUnitTestCase<LaunchActivity> {
    ...

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mLaunchIntent = new Intent(getInstrumentation()
                .getTargetContext(), LaunchActivity.class);
        startActivity(mLaunchIntent, null, null);
        final Button launchNextButton =
                (Button) getActivity()
                .findViewById(R.id.launch_next_activity_button);
    }
}
```

验证另一个Activity的启动

我们的单元测试目标可能包括:

- 验证当Button被按下时，启动的LaunchActivity是否正确。
- 验证启动的Intent是否包含有效的数据。

为了验证一个触发Intent的Button的事件，我们可以使用getStartedActivityIntent()方法。通过使用断言方法，我们可以验证返回的Intent是否为空，以及是否包含了预期的数据来启动下一个Activity。如果两个断言值都是真，那么我们就成功地验证了Activity发送的Intent是正确的了。

我们可以这样实现测试方法:

```
@MediumTest
public void testNextActivityWasLaunchedWithIntent() {
    startActivity(mLaunchIntent, null, null);
    final Button launchNextButton =
            (Button) getActivity()
            .findViewById(R.id.launch_next_activity_button);
    launchNextButton.performClick();

    final Intent launchIntent = getStartedActivityIntent();
    assertNotNull("Intent was null", launchIntent);
    assertTrue(isFinishCalled());

    final String payload =
            launchIntent.getStringExtra(NextActivity.EXTRAS_PAYLOAD_KEY);
    assertEquals("Payload is empty", LaunchActivity.STRING_PAYLOAD, payload);
}
```

因为LaunchActivity是独立运行的，所以不可以使用TouchUtils库来操作UI。如果要直接进行Button点击，我们可以调用perfoemClick()方法。

本节示例代码AndroidTestingFun.zip

创建功能测试

> 编写:huanglizhuo - 原文:http://developer.android.com/training/activity-testing/activity-functional-testing.html

功能测试包括验证单个应用中的各个组件是否与使用者期望的那样（与其它组件）协同工作。比如，我们可以创建一个功能测试验证在用户执行UI交互时Activity是否正确启动目标Activity。

要为Activity创建功能测，我们的测试类应该对ActivityInstrumentationTestCase2进行扩展。与ActivityUnitTestCase不同，ActivityInstrumentationTestCase2中的测试可以与Android系统通信，发送键盘输入及点击事件到UI中。

要了解一个完整的测试例子可以参考示例应用中的 `SenderActivityTest.java` 。

添加测试方法验证函数的行为

我们的函数测试目标应该包括:

- 验证UI控制是否正确启动了目标Activity。
- 验证目标Activity的表现是否按照发送Activity提供的数据呈现。

我们可以这样实现测试方法:

```
@MediumTest
public void testSendMessageToReceiverActivity() {
    final Button sendToReceiverButton = (Button)
            mSenderActivity.findViewById(R.id.send_message_button);

    final EditText senderMessageEditText = (EditText)
            mSenderActivity.findViewById(R.id.message_input_edit_text);

    // Set up an ActivityMonitor
    ...

    // Send string input value
    ...

    // Validate that ReceiverActivity is started
    ...

    // Validate that ReceiverActivity has the correct data
    ...

    // Remove the ActivityMonitor
    ...
}
```

测试会等待匹配的Activity启动，如果超时则会返回null。如果ReceiverActivity启动了，那么先前配置的ActivityMoniter就会收到一次碰撞（Hit）。我们可以使用断言方法验证ReceiverActivity是否的确启动了，以及ActivityMoniter记录的碰撞次数是否按照预想地那样增加。

设立一个ActivityMonitor

为了在应用中监视单个Activity我们可以注册一个ActivityMoniter。每当一个符合要求的Activity启动时，系统会通知ActivityMoniter，进而更新碰撞数目。

通常来说要使用ActivityMoniter，我们可以这样：

1. 使用getInstrumentation()方法为测试用例实现Instrumentation。
2. 使用Instrumentation的一种addMonitor()方法为当前instrumentation添加一个Instrumentation.ActivityMonitor实例。匹配规则可以通过IntentFilter或者类名字符串。
3. 等待开启一个Activity。
4. 验证监视器撞击次数的增加。
5. 移除监视器。

下面是一个例子:

```
// Set up an ActivityMonitor
ActivityMonitor receiverActivityMonitor =
        getInstrumentation().addMonitor(ReceiverActivity.class.getName(),
        null, false);

// Validate that ReceiverActivity is started
TouchUtils.clickView(this, sendToReceiverButton);
ReceiverActivity receiverActivity = (ReceiverActivity)
        receiverActivityMonitor.waitForActivityWithTimeout(TIMEOUT_IN_MS);
assertNotNull("ReceiverActivity is null", receiverActivity);
assertEquals("Monitor for ReceiverActivity has not been called",
        1, receiverActivityMonitor.getHits());
assertEquals("Activity is of wrong type",
        ReceiverActivity.class, receiverActivity.getClass());

// Remove the ActivityMonitor
getInstrumentation().removeMonitor(receiverActivityMonitor);
```

使用Instrumentation发送一个键盘输入

如果Activity有一个EditText，我们可以测试用户是否可以给EditText对象输入数值。

通常在ActivityInstrumentationTestCase2中给EditText对象发送串字符，我们可以这样做：

1. 使用runOnMainSync()方法在一个循环中同步地调用requestFocus()。这样，我们的UI线程就会在获得焦点前一直被阻塞。
2. 调用waitForIdleSync()方法等待主线程空闲（也就是说,没有更多事件需要处理）。
3. 调用sendStringSync()方法给EditText对象发送一个我们输入的字符串。

比如:

```
// Send string input value
getInstrumentation().runOnMainSync(new Runnable() {
    @Override
    public void run() {
        senderMessageEditText.requestFocus();
    }
});
getInstrumentation().waitForIdleSync();
getInstrumentation().sendStringSync("Hello Android!");
getInstrumentation().waitForIdleSync();
```

本节例子AndroidTestingFun.zip